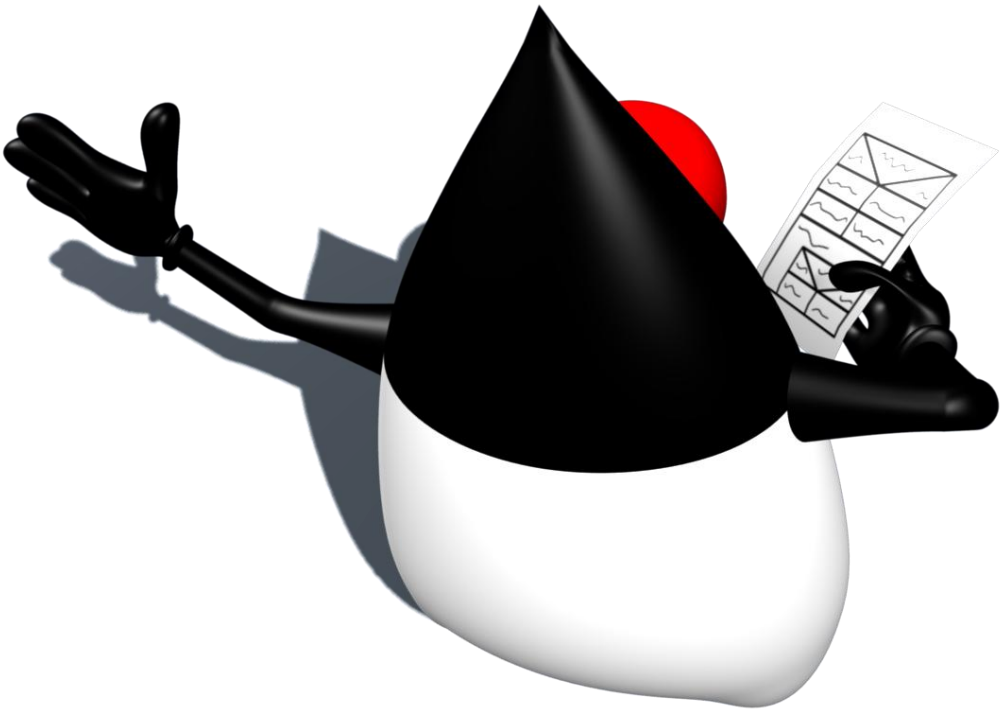
De Java Applet Creator

Profielwerkstuk

Willem Sonke, G6C



Inhoudsopgave

Inhoudsopgave **2**

Inleiding **5**

Stap 1: Probleem definiëren **6**
Ontwerp van de GUI 7
Blokken toevoegen en verplaatsen 7
Een blok bewerken 9
Methoden 10
Uitvoeren naar Java 11
Welke programmeertaal? 11
Applet of Application? 12
Randvoorwaarden 12

Stap 2: Probleem analyseren **13**
Code en blokken 13
Standaardcode 13
Benodigde blokken 14
Herhalingsstructuren 14
Controlestructuren 15
Methoden 15
Variabelen 15
Tekenen 16
GUI 17
JavaLogo 18

De PSD Creator **19**
Stap 1: Probleem definiëren 19
Stap 2: Probleem analyseren 20
Tekenen 20
Verschuiven 22
Stap 3: Oplossing schematiseren 22
Tekenen 22
Verschuiven 24
Selecteren 25
Stap 4: Programma-instructies coderen 25
Klassenindeling 25
Klasse PSDCreator 25
Klasse PSD 26
Methode *bepaalBlokLengte* 26
Methoden *verschuifBlokOmlaag* en *verschuifBlokOmhoog* 27
Methode *zoekGekliktBlok* 28
Klasse ActieAfhandelaar 29
Klasse Blok 29
Klasse Werkbalk 29
Stap 5: Testen 29
Ingewikkelde PSD's testen 30
Een leeg herhalings- of keuzeblok is niet duidelijk 31
Een pijltje in het keuzeblok bij een lege kolom 32
Specifieke probleemgevallen testen 32
Blokken schuiven 33

Blokken verwijderen 34
Blokken selecteren in ingewikkelde PSD's 34
Stap 6: Documenteren 34
De code 34
Gebruikershandleiding 34
Stap 2: Probleem analyseren (vervolg) 35
Algoritmen voor het PSD 35
Uitwerking van de benodigde blokken 36
Het uitvoeralgoritme 37
Stap 3: Oplossing schematiseren 40
Het uitvoeralgoritme 40
genereerCode() 40
genereerCodeInitialisatie() 41
genereerCodeTekenmethode() 41
genereerCodeAndereMethoden() 41
schrijfNaarVector(Vector, String) 42
vectorSchrijven(Vector) 42
schrijf(String) 42
Stap 4: Programma-instructies coderen 43
De klassenindeling 43
Klassen voor de GUI (Graphical User Interface) 43
Klassen voor de opslag van het PSD 44
Klassen voor de uitvoer 45
De GUI maken 45
Klasse AppletCreator 45
Klasse HoofdMenubalk 45
Klasse HoofdContentPane 46
Klasse HoofdStatusbalk 46
Klasse Methodenlijst 47
Klasse PSDGebied 47
Klasse PSDPaneel 47
Klasse PSDWerkbalk 49
Klasse Blokkentabs 50
Klasse BlokInTabs 50
Klasse ActieAfhandelaar 51
Klasse BlokBewerkenVenster 51
Klasse MethodeBewerkenVenster 52
Klasse StandaardMethodeBewerkenVenster 54
Klasse Aboutbox 55
Klasse Helpvenster 55
Klasse OptiesVenster 57
De opslag van het PSD maken 57
Klasse MethodeVerzameling 57
Interface Methode 58
Klasse InitialisatieMethode 58
Klasse TekenMethode 59
Klasse AndereMethode 59
Klasse Blok 59
De uitvoer 60

Klasse UitvoerModule ... 60
Klasse HTMLModule ... 61
Aanpassingen aan ActieAfhandelaar ... 62
Openen en opslaan ... 64

Stap 5: Testen ... 65
De Mondriaan-applet ... 65
Probleem: blok wordt niet ververst bij sluiten BlokBewerkenVenster ... 67
Huisjes en methoden ... 67
Probleem: het Graphics-object in eigen methoden krijgen ... 68
Probleem: fouten bij methodenheaders van eigen methoden ... 70
Applet met een GUI ... 71
De Reken-applet ... 72
De Reisadvies-applet ... 76
Concentrische cirkels ... 78
De Priem-applet ... 80

Stap 6: Documenteren ... 83
De helpbestanden schrijven ... 83
De handleiding ... 85
JAR-bestand maken ... 86
De website maken ... 87
Javadoc-documentatie genereren ... 88

Overzicht van de bestanden op de CD ... 89

Evaluatie ... 90

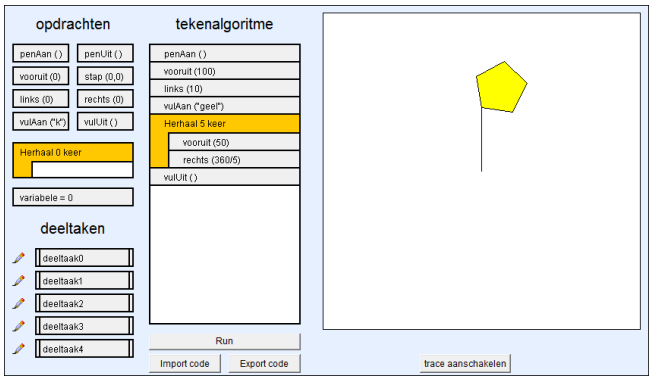
Bronnenlijst ... 91

Inleiding

Programmeertalen zijn vaak lastig om te leren, vooral voor mensen die nog nooit eerder hebben geprogrammeerd. Toch denk ik dat er mensen zijn die graag een klein programmaatje zouden willen schrijven. Daardoor kwam ik op het idee van de Java Applet Creator: een programma waarmee gebruikers op een relatief eenvoudige manier een kleine applet in elkaar kunnen zetten. De gebruiker maakt op het scherm een PSD aan van de dingen die er moeten gebeuren en de Applet Creator zorgt voor alle code. Op die manier zouden gebruikers die niets van programmeren afweten toch relatief gemakkelijk een programma moeten kunnen schrijven; ze hoeven alleen iets te leren over PSD's en erg moeilijk is dat niet.

Er zijn al andere programma's op internet met ongeveer hetzelfde doel. Een voorbeeld daarvan is ArgoUML[1], dat echter geen PSD's maakt, maar diagrammen in UML. Met ArgoUML is het mogelijk om C++- en Java-code te genereren. De inhoud van de methoden blijft echter leeg, omdat er geen voorgedefinieerde functies in ArgoUML ingebouwd zijn.
Een ander voorbeeld is de NSD-Editor[2], die is geschreven in Delphi. Met dit programma, dat voornamelijk bedoeld is om PSD's te maken, kan ook broncode in de programmeertalen C en Pascal gemaakt worden. Ook hierbij geldt dat de methoden leeg blijven.

Een derde voorbeeld is de applet van Informatica-Actief[3]. Zie de screenshot in *Figuur 1*. Deze werkt wel in Java, maar genereert pseudo-code in plaats van Java-code. Deze code is bovendien beperkt tot JavaLogo.



Figuur 1 – De applet over algoritmen van Informatica-Actief

Wat ik wil gaan doen, dus volledige programma's maken aan de hand van een PSD met voorgedefinieerde blokken, heb ik nergens op internet gevonden.

De Java Applet Creator ga ik maken volgens de zes stappen van programmaontwikkeling:

1. *Probleem definiëren*: bedenken wat het programma moet kunnen
2. *Probleem analyseren*: vaststellen hoe je de resultaten krijgt (algoritmen bedenken)
3. *Oplossing schematiseren*: PSD's tekenen van de algoritmen
4. *Programma-instructies coderen*: het eigenlijke programmeerwerk
5. *Testen*: fouten (vooral semantische) opsporen en verhelpen
6. *Documenteren*: dit gebeurt eigenlijk door het hele proces heen. In deze stap zal ik dus vooral de gebruikershandleiding schrijven en commentaar in de code opnemen

Deze zes stappen zullen in dit verslag terug komen als hoofdstukken.

[1] Zie *http://argouml.tigris.org/*
[2] Zie *http://diuf.unifr.ch/softeng/student-projects/completed/kalt/NSD.html*
[3] Deze kan alleen worden gebruikt met gebruikersnaam en wachtwoord van de site van Informatica-Actief.

Stap 1: Probleem definiëren

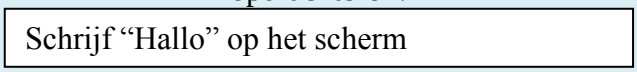
Als eerste heb ik bedacht wat het programma moet kunnen. Uiteindelijk ben ik tot deze doelstelling gekomen: *De Java Applet Creator moet minstens alle applets uit het boekje* Java: my cup of tea *kunnen maken, op zo'n manier dat ze ook gemaakt kunnen worden door mensen die niets van programmeren afweten.* Dit betekent dat de applets die de Applet Creator maakt, minstens het volgende moeten kunnen:

- Tekenen naar het scherm m.b.v. het Graphics-object
- Berekeningen maken met eenvoudige operatoren
- Uit meerdere methoden bestaan, wel uit slechts één klasse
- Buttons, TextFields en Labels gebruiken, inclusief afhandelen van acties
- Verschillende Fonts gebruiken voor deze elementen
- Werken met strings en strings inlezen uit TextFields
- Controle- en herhalingsstructuren gebruiken

Het programma zal werken met PSD's (zie het kader hieronder). De gebruiker kan een PSD maken en de Applet Creator maakt dan automatisch code aan. In het vervolg van deze stap zal ik laten zien hoe ik het beste een GUI (Graphical User Interface) kan maken die ervoor zorgt dat aan deze eisen voldaan wordt.

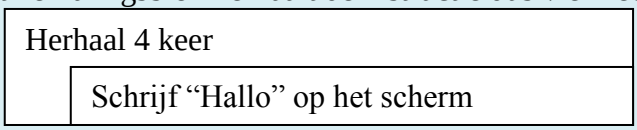
PSD's

Een PSD (ProgrammaStructuurDiagram) of NSD (Nassi-Shneidermandiagram) is een simpel diagram waarmee het verloop van een programma wordt aangegeven. De basis van zo'n diagram is een opdrachtblok: een rechthoek waarin een opdracht wordt geschreven (gewoon in het Nederlands, dus niet in een programmeertaal). Hieronder staat een voorbeeld van een opdrachtblok.

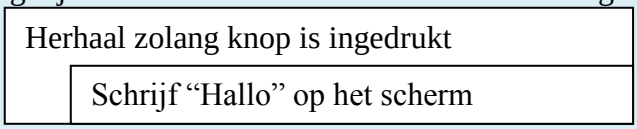


Je kunt meerdere opdrachtblokken onder elkaar zetten, die worden dan na elkaar uitgevoerd.

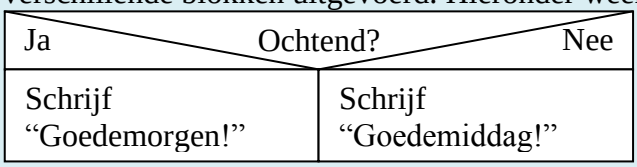
Er zijn nog twee speciale blokken: het herhalingsblok en het keuzeblok. Een herhalingsblok is een blok waarmee opdrachtblokken kunnen worden herhaald. Een voorbeeld staat hieronder; dit herhalingsblok herhaalt de instructie dus vier keer.



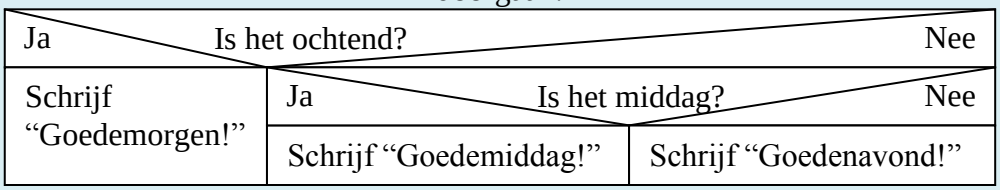
Het is ook mogelijk om andere voorwaarden in een herhalingsblok te zetten:



Het tweede speciale blok is een keuzeblok. Hiermee wordt een keuze gemaakt, en afhankelijk hiervan worden verschillende blokken uitgevoerd. Hieronder weer een voorbeeld.



In plaats van opdrachtblokken kun je in deze twee soorten speciale blokken ook weer andere speciale blokken zetten. Uiteraard kunnen in die blokken dan ook weer andere blokken gezet worden. De nesting (het in elkaar zetten) van blokken kan in principe zelfs oneindig doorgaan.

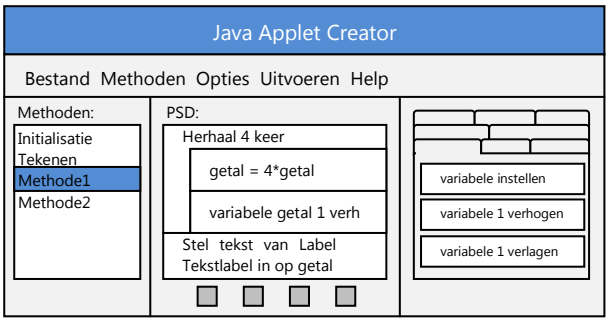


Met een PSD kan bijna ieder programma beschreven worden. Dat is het mooie van PSD's: een heel simpel diagram kan bijna alle structuren van het programmeren beschrijven.

Ontwerp van de GUI

In het hoofdscherm van de Applet Creator staat aan de bovenkant een menubalk met de volgende menu's:

- **Bestand**, hierin kan de gebruiker een Applet Creator-project openen en opslaan, een nieuw project maken en de Applet Creator afsluiten.
- **Methoden**, hiermee kan de gebruiker methoden maken, wijzigen en verwijderen.
- **Opties**, hiermee kan de gebruiker enkele opties voor de Applet Creator instellen.
- **Uitvoeren**, hiermee kan de gebruiker het project uitvoeren naar Java-code en, indien dit mogelijk blijkt te zijn tijdens het programmeren van de Applet Creator, deze code ook direct compileren en uitvoeren.
- **Help**, hiermee kan de gebruiker help opvragen over het programma.



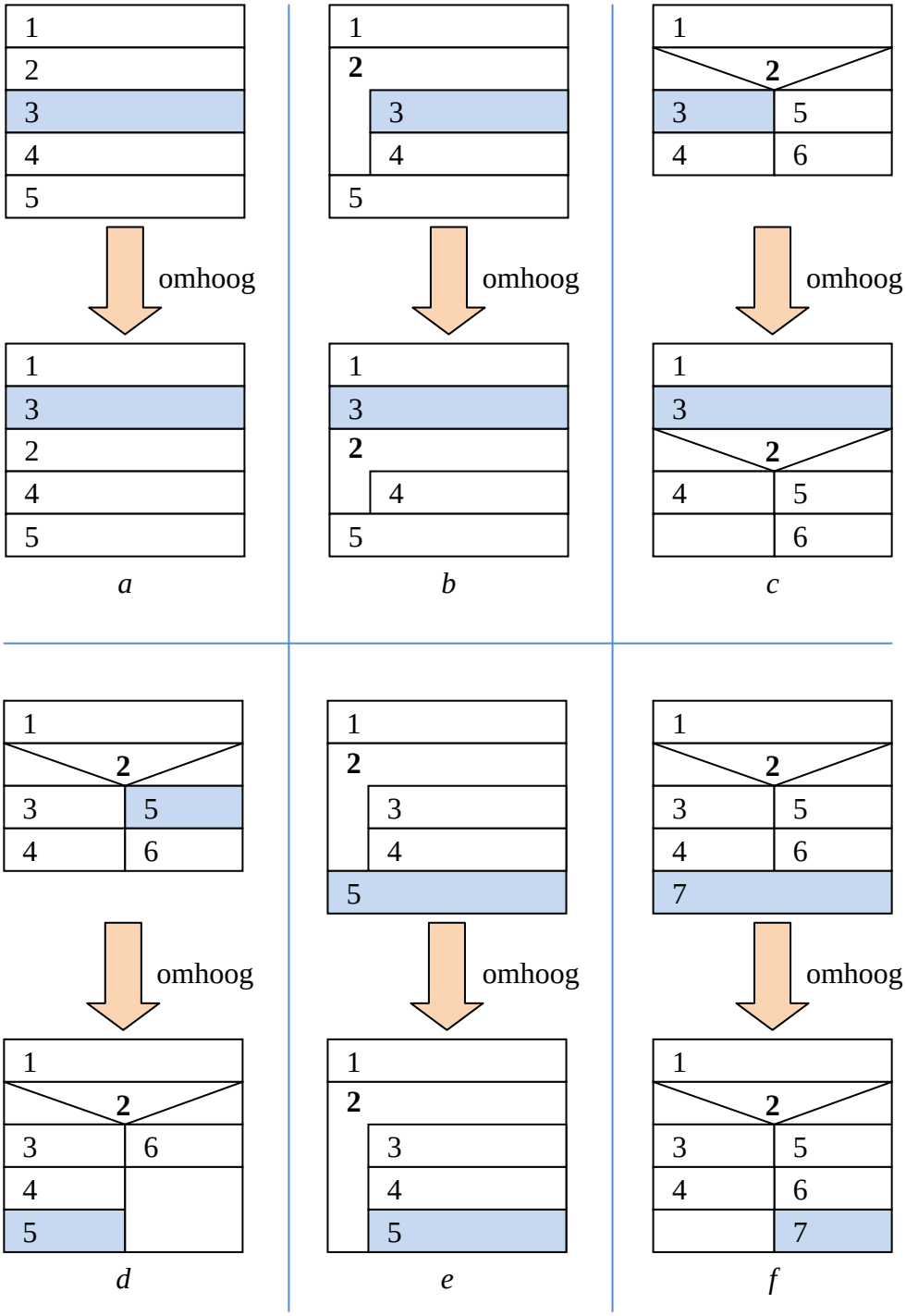
Figuur 2 – Basisontwerp voor de GUI

De rest van het scherm is verdeeld in drie gebieden (*Figuur 2*). Aan de linkerkant van het scherm staat een lijst van alle methoden die de gebruiker heeft aangemaakt. (Een methode aanmaken kan via het Methoden-menu, zie het kopje Methoden hieronder.) Als de gebruiker klikt op een methode, wordt deze geselecteerd en het PSD dat bij deze methode hoort, verschijnt in het midden van het scherm, dat het PSD-gebied heet. Onder het PSD staan knopjes om een blok te verplaatsen, te bewerken en te verwijderen. Aan de rechterkant van het scherm staat een serie tabbladen met daarin beschikbare blokken die gebruikt kunnen worden in het PSD. De gebruiker kan blokken vanuit de tabs verplaatsen naar het PSD (zie het kopje Blokken toevoegen en verplaatsen).

Blokken toevoegen en verplaatsen

Mijn eerste idee over het toevoegen en verplaatsen van blokken was een sleep-interface: de gebruiker kan blokken uit de tabs en uit het PSD gewoon slepen naar een andere plaats. Het lijkt mij echter dat dat zeer ingewikkeld te programmeren is, omdat het al moeilijk te bepalen zou zijn waar de muis staat, en bovendien moet er dan ook nog worden bepaald op welke positie in het PSD het blok moet worden neergezet. Dat lijkt niet zo moeilijk, maar denk eens aan de onderkant van een herhalingsblok; hoe moet je bepalen of de gebruiker het blok onder

het herhalingsblok wil hebben of juist onder in het herhalingsblok? Deze moeilijkheden hebben me ertoe gebracht om een andere interface te bedenken die niet met slepen werkt.
Als de gebruiker op een blok in de tabs klikt, dan verschijnt eerst het venster om de gegevens voor het blok in te stellen (zie het kopje Een blok bewerken). Hierna wordt het blok achteraan in het PSD toegevoegd. Het wordt direct geselecteerd.
Als een gebruiker op een blok in het PSD klikt, dan wordt dit blok ook geselecteerd. Dit is te zien doordat het blok een andere kleur krijgt. Onder het PSD staan twee knopjes om het geselecteerde blok te verplaatsen:



Figuur 3 – Wat er gebeurt als er op "omhoog" wordt geklikt. Het blauwe blok is steeds geselecteerd.

- **omhoog** – het geselecteerde blok één omhoog verplaatsen. Zie *Figuur 3* voor een overzicht van de werking van de omhoog-knop.
    - Als er zich boven het geselecteerde blok een opdrachtblok bevindt, dan wisselen dit blok en het geselecteerde blok van plaats (*Figuur 3a*).
    - Als het geselecteerde blok als bovenste in een herhalingsblok zit, dan schuift het geselecteerde blok uit het herhalingsblok (*Figuur 3b*).
    - Als het geselecteerde blok als bovenste in de "ja"-kolom van een keuzeblok zit, dan schuift het geselecteerde blok uit het keuzeblok (*Figuur 3c*).
    - Als het geselecteerde blok als bovenste in de "nee"-kolom van een keuzeblok zit, dan schuift het geselecteerde blok door naar de onderkant van de "ja"-kolom (*Figuur 3d*).
    - Als er zich boven het geselecteerde blok een herhalingsblok bevindt, dan schuift het geselecteerde blok onderin het herhalingsblok (*Figuur 3e*).
    - Als er zich boven het geselecteerde blok een keuzeblok bevindt, dan schuift het geselecteerde blok onderin de "nee"-kolom van het keuzeblok (*Figuur 3f*).
- **omlaag** – het geselecteerde blok één omlaag verplaatsen. Aangezien *omlaag* precies het tegenovergestelde doet als *omhoog* is de werking hier te zien door *Figuur 3* "om te draaien". In plaats van de figuur van boven naar onderen te lezen, kan hij dan van onderen naar boven gelezen worden. De volgorde is nu wel iets anders, zie de verwijzingen.
    - Als er zich onder het geselecteerde blok een opdrachtblok bevindt, dan wisselen dit blok en het geselecteerde blok van plaats (*Figuur 3a* omgekeerd).
    - Als het geselecteerde blok als onderste in een herhalingsblok zit, dan schuift het geselecteerde blok uit het herhalingsblok (*Figuur 3e* omgekeerd).
    - Als het geselecteerde blok als onderste in de "nee"-kolom van een keuzeblok zit, dan schuift het geselecteerde blok uit het keuzeblok (*Figuur 3f* omgekeerd).
    - Als het geselecteerde blok als onderste in de "ja"-kolom van een keuzeblok zit, dan schuift het geselecteerde blok door naar de bovenkant van de "nee"-kolom (*Figuur 3d* omgekeerd).
    - Als er zich onder het geselecteerde blok een herhalingsblok bevindt, dan schuift het geselecteerde blok in het bovenherhalingsblok (*Figuur 3b* omgekeerd).
    - Als er zich onder het geselecteerde blok een keuzeblok bevindt, dan schuift het geselecteerde blok bovenin de "ja"-kolom van het keuzeblok (*Figuur 3c* omgekeerd).

Verder is er nog een knop **Blok verwijderen** die het geselecteerde blok verwijdert uit het PSD.

Een blok bewerken

Als de gebruiker op het knopje Blok bewerken klikt of een nieuw blok neerzet, dan verschijnt er een venster waarin opties van het blok kunnen worden aangepast (*Figuur 4*). Bijvoorbeeld bij een herhalingsblok: *Hoeveel keer herhalen*, of bij een blok om een variabele in te stellen: *Welke variabele* en *Op welke waarde moet deze worden ingesteld*.

Figuur 4 – Ontwerp voor het bewerkvenster

Figuur 5 – Ontwerp voor het methodevenster

Methoden

Als de gebruiker in het menu Methoden een nieuwe methode aanmaakt of de huidige aanpast, verschijnt een scherm om de eigenschappen van de methode te bewerken (*Figuur 5*). De gebruiker kan hierin ingeven:

- de naam van de methode;
- is de methode public of private;
- wat zijn de parameters (in tabelvorm);
- wat zijn de gebruikte variabelen (in tabelvorm);
- wat is de resultaatwaarde (void of een variabeletype).

Standaard staan er in ieder project al twee methoden: *Initialisatie* en *Tekenen*.

- *Initialisatie* komt ongeveer overeen met `init`. Dit is dus de methode die wordt uitgevoerd als de applet wordt gestart.
- *Tekenen* komt ongeveer overeen met `paint`. Deze methode staat apart omdat de applet dan met `repaint()` gemakkelijk opnieuw kan worden getekend.
- De `action`-methode is geen aparte methode van de Applet Creator. Deze wordt automatisch gegenereerd aan de hand van de informatie die wordt gegeven bij het maken van de GUI. Hierbij kan de gebruiker namelijk aangeven welke methode er moet worden geactiveerd bij een actie.

Bij deze standaardmethoden verschijnt er, als er op Wijzigen wordt geklikt, een aangepast dialoogvenster waarin alleen de varibelen kunnen worden ingegeven.

`init`, `paint`, `action`

In een Java-applet zijn er drie “standaard”-methoden: `init`, `paint` en `action`.

De methode `init` wordt uitgevoerd bij het starten van de applet. Hierin wordt de GUI gemaakt en kan bijvoorbeeld de achtergrondkleur van de applet worden ingesteld.

De methode `paint` wordt uitgevoerd na `init`. In deze methode kunnen het beste de aanroepen van het Graphics-object staan, dus tekenopdrachten.

De methode `action` bevat de afhandeling van acties, bijvoorbeeld wat er gebeurt als de gebruiker van de applet op een knop klikt.

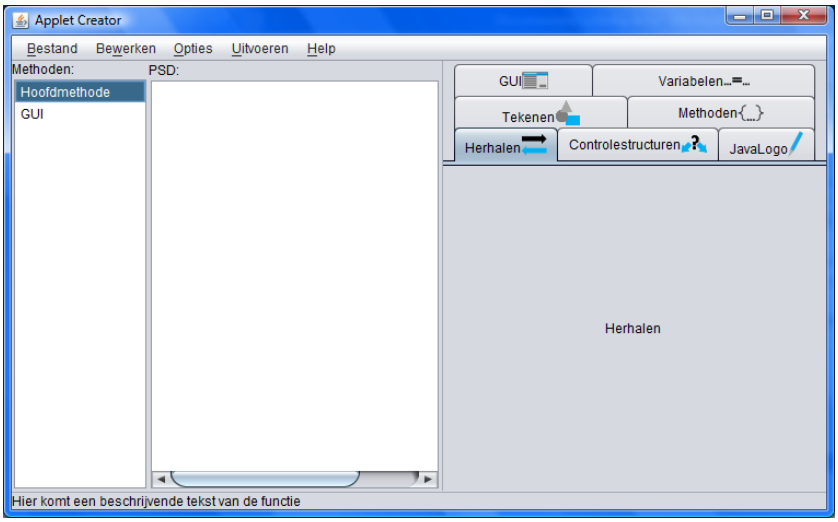
Uitvoeren naar Java

In het menu Uitvoeren staan er één of twee opties:

- In ieder geval de optie *Uitvoeren naar Java*, deze knop activeert een venstertje waarin de gebruiker kan kiezen waar de .java-file moet worden opgeslagen. Als de gebruiker dat heeft ingegeven, verschijnt er een scherm met een voortgangsindicator, die aangeeft hoelang het uitvoeren nog gaat duren. (Het genereren van de code kan pas ná het opgeven van de bestandsnaam, want de naam van de klasse moet overeenkomen met de bestandsnaam. Bijvoorbeeld in *MijnApplet.java* mag alleen de klasse *MijnApplet* worden gedefinieerd.) Als de uitvoer gereed is, verschijnt er een melding en daarna komt de gebruiker weer terug in het hoofdscherm. De gebruiker kan de code bekijken met bijvoorbeeld Kladblok.
- De tweede optie is Uitvoeren en runnen. Dit doet hetzelfde als de eerste optie, maar deze optie compileert direct de Java-code tot een .class-file en voert deze ook direct uit. In het opties-scherm kan de gebruiker dan instellen waar de JDK en de JRE zich bevinden. Deze optie komt er misschien niet; dit hangt ervanaf of ik vanuit Java de compiler en de JRE kan aansturen, ik weet namelijk nog niet of dat mogelijk is. Deze functie zou wel heel handig zijn voor gebruikers, kijk maar naar de doelstelling: *… op zo'n manier dat ze ook gemaakt kunnen worden door mensen die niets van programmeren afweten*. Als je eerst nog met de hand moet compileren en runnen moet je natuurlijk wel weten hoe dat moet en dat is vervelend voor gebruikers die niets van programmeren afweten en dan in de Help-functie moeten gaan zoeken hoe dat moet.

Welke programmeertaal?

Mijn eerste plan was om als programmeertaal Java te gebruiken, omdat ik daar nog niet zoveel ervaring mee had en het me leuk leek om een programmeertaal te gebruiken die ik niet zo goed kende. Vanwege de omvang van de opdracht leek het me later echter beter om Delphi te gebruiken, aangezien ik al eerder een opdracht in deze programmeertaal gemaakt had. Ik heb toen besloten om eens te onderzoeken of het in Java makkelijk is om een GUI te maken; hiervoor heb ik een eenvoudige versie van de GUI van de Applet Creator in Java gemaakt (*Figuur 6*). Alhoewel dit lang duurde, heb ik nu wel het idee dat ik begrijp hoe je in Java een groot programma maakt, dus ik heb besloten om de opdracht toch in Java te maken.



Figuur 6 – Schermafdruk van de proef-GUI

Applet of Application?

De tweede vraag is: ga ik een applet maken of een Application? De verschillen zijn als volgt:

- Een applet draait op een webpagina, een Application is een losstaand programma.
- Een applet kan niet naar de harde schijf schrijven, een Application wel.
- Applications kunnen waarschijnlijk niet met JavaLogo werken, applets wel.

Uiteindelijk ben ik tot de conclusie gekomen om een Application te maken, vooral vanwege de reden dat een applet niet kan openen en opslaan. Bovendien zal een applet zeker geen compiler kunnen aanroepen om de code te compileren.

Randvoorwaarden

- Tijd: ik verwacht dat ik qua tijd meer nodig heb dan de 80 uur die er staan voor het profielwerkstuk.
- Hardware: ik werk thuis op de computer en op school in de infotheek. Waarschijnlijk werk ik niet tijdens de informaticales, omdat we daar met een ander project bezig zijn.
- Software: ik gebruik de JDK versie 6 update 17 voor dit project. Verder heb ik Kladblok nodig (voor het maken van de broncode), Word (voor het schrijven van deze documentatie) en Excel (voor het maken van de planning).
- Een goede planning.

Stap 2: Probleem analyseren

In deze stap wordt de bespreking van het probleem uitgediept. Bovendien ga ik kijken hoe het probleem in grote lijnen kan worden opgelost.

Code en blokken

Als eerste ga ik kijken welke standaardcode er in iedere applet gemaakt moet worden. Hierna zal ik bespreken welke blokken er nodig zijn (en welke code die genereren). Ten slotte zal ik een algoritme voor de uitvoer bespreken.

Standaardcode

De standaardcode voor iedere applet is:

```
import java.applet.*; import java.awt.*;

public class Klassenaam extends Applet
{
    VariabeleDeclaratieVanGUI

    public void init()
    {
        VariabelenInit
        InstructiesInit
    }

    public void paint(Graphics pen)
    {
        VariabelenPaint
        InstructiesPaint
    }

    public boolean action(Event e, Object o)
    {
        InstructiesAction
        return false;
    }

    OverigeMethoden
}
```

Hierin moeten de volgende dingen ingevuld worden.

- *Klassenaam*: deze geeft de gebruiker impliciet in door de bestandsnaam van het uitvoerbestand op te geven. Bijvoorbeeld: als de gebruiker het bestand MijnApplet.java laat maken, dan wordt *Klassenaam* MijnApplet.
- *VariabeleDeclaratieVanGUI*: deze wordt gevuld door de GUI-blokken. Bijvoorbeeld het blok *Maak een knop (Button)* maakt ook hier een instructie aan. Zie hiervoor de GUI-blokken in de paragraaf *Benodigde blokken*.
- *VariabelenInit*: de reeks van variabelen uit het methodevenster van Initialiseren; steeds op de volgende manier: *Soort Variabelenaam;*
- *InstructiesInit*: dit is een reeks instructies die wordt gegeven door de blokken in de methode Initialiseren. Bij het uitvoerproces wordt voor ieder blok in Initialiseren de code genomen (zie paragraaf *Benodigde blokken* voor deze code per blok) en daarna wordt dit achter elkaar in *InstructiesInit* gezet.
- *VariabelenPaint*: de reeks van variabelen uit het methodevenster van Tekenen.

- *InstructiesPaint*: hetzelfde als *InstructiesInit* voor alle blokken in Tekenen.
- *InstructiesAction*: deze wordt weer gevuld door de GUI-blokken. Zie hiervoor de GUI-blokken in de paragraaf *Benodigde blokken*.
- *OverigeMethoden*: hier worden alle andere methoden neergezet volgens onderstaand schema.

```
Toegang Resultaatwaarde Methodenaam(Parameters)
{
    VariabelenDeclaratie
    InstructiesMethode
}
```

Hierin moeten weer dingen ingevuld worden:

- *Toegang*: *public* of *private*, dit wordt ingesteld in het methodescherm.
- *Resultaatwaarde*: *init*, *double*, *String* of *void*, dit wordt ingesteld in het methodescherm.
- *Methodenaam*: de naam van de methode, ingevuld in het methodescherm.
- *Parameters*: de parameters van de methode. Bijvoorbeeld *String naam, double massa*. Deze worden weer ingesteld in het methodescherm.
- *VariabelenDeclaratie*: de variabelen van de methode. Ook deze worden ingevoerd in het methodescherm.
- *InstructiesMethode*: deze worden gevuld door de blokken in de methode.

Benodigde blokken

Hieronder staan de blokken die ik denk nodig te hebben, gegroepeerd per tabblad. De **dikgedrukte** naam is de naam van het blok zoals die in de tabbladen staat. Verder staat er bij ieder blok de code die er moet gegenereerd bij de uitvoer van het programma.

Herhalingsstructuren

- **Herhaal een bepaald aantal keer**
  Dit blok heeft een speciale vorm, op die manier kan het blok andere blokken bevatten (de *blokInhoud*)
  Opties die gegeven kunnen worden:
  - Hoeveel keer herhalen? (*aantalKeer*)
  - In welke variabele moet de teller worden bijgehouden? (*tellerVariabele*)

  De gegenereerde code is:

```
for (tellerVariabele = 0; tellerVariabele <= aantalKeer;
tellerVariabele++)
{
    blokInhoud
}
```

- **Herhaal zolang een voorwaarde waar is**
  Dit blok heeft een speciale vorm, op die manier kan het blok andere blokken bevatten (de *blokInhoud*)
  Opties die gegeven kunnen worden:
  - Welke voorwaarde? (*voorwaarde*)

  De gegenereerde code is:

```
while (voorwaarde)
{
```

```
    blokInhoud
}
```

Controlestructuren

- **Keuze op basis van een vraag**
  Dit blok heeft een special vorm, op die manier kan het blok op twee plekken andere blokken bevatten, namelijk bij de kolom Ja (*blokInhoudJa*) en de kolom Nee (*blokInhoudNee*).
  Opties die gegeven kunnen worden:
  - Welke vraag? (*vraag*)

  De gegenereerde code is:

```
if (vraag) {
    blokInhoudJa
} else {
    blokInhoudNee
}
```

Methoden

- **Voer een methode uit**
  Opties die gegeven kunnen worden:
  - Welke methode? (*methode*)
  - Welke parameters moeten er met deze methode worden meegegeven? (*parameters*)

  De gegenereerde code is:

```
this.methode(parameters);
```

- **Een waarde teruggeven**
  Opties die gegeven kunnen worden:
  - Welke waarde moet worden teruggegeven? Let wel op: hij moet van het type zijn zoals beschreven in het methode-bewerken-venster. (*waarde*)

  De gegenereerde code is:

```
return waarde;
```

Variabelen

- **Stel een variabele in**
  Opties die gegeven kunnen worden:
  - Welke variabele? (*variabele*)
  - Welke waarde? (*waarde*)

  De gegenereerde code is:

```
variabele = waarde;
```

- **Verhoog een variabele met 1**
  Opties die gegeven kunnen worden:
  - Welke variabele? (*variabele*)

  De gegenereerde code is:

```
variabele++;
```

- **Verlaag een variabele met 1**
  Opties die gegeven kunnen worden:
  - Welke variabele? (*variabele*)

De gegenereerde code is:

```
variabele--;
```

Tekenen

- **Schrijf tekst op het scherm**
  Opties die gegeven kunnen worden:
  - Welke tekst? Let op: als je letterlijke tekst wilt schrijven, zet er dan dubbele aanhalingstekens omheen. Wil je een variabele schrijven, zet er dan geen aanhalingstekens omheen. Stukken tekst kun je aan elkaar "lijmen" met een plusteken. (*tekst*)
  - Wat is de x-coördinaat? (*xCoord*)
  - Wat is de y-coördinaat? (*yCoord*)

  Gegenereerde code:

```
pen.drawString(tekst, xCoord, yCoord);
```

- **Teken een lijn**
  Opties die gegeven kunnen worden:
  - x-coördinaat beginpunt? (*xBegin*)
  - y-coördinaat beginpunt? (*yBegin*)
  - x-coördinaat eindpunt? (*xEind*)
  - y-coördinaat eindpunt? (*yEind*)

  Gegenereerde code:

```
pen.drawLine(xBegin, yBegin, xEind, yEind);
```

- **Teken een rechthoek**
  Opties die gegeven kunnen worden:
  - x-coördinaat linksbovenhoek? (*xCoord*)
  - y-coördinaat linksbovenhoek? (*yCoord*)
  - Breedte? (*breedte*)
  - Hoogte? (*hoogte*)

  Gegenereerde code:

```
pen.drawRect(xCoord, yCoord, breedte, hoogte);
```

- **Teken een gevulde rechthoek**
  Opties die gegeven kunnen worden:
  - x-coördinaat linksbovenhoek? (*xCoord*)
  - y-coördinaat linksbovenhoek? (*yCoord*)
  - Breedte? (*breedte*)
  - Hoogte? (*hoogte*)

  Gegenereerde code:

```
pen.fillRect(xCoord, yCoord, breedte, hoogte);
```

- **Teken een ovaal**
  Opties die gegeven kunnen worden:
  - x-coördinaat linksbovenhoek? (*xCoord*)
  - y-coördinaat linksbovenhoek? (*yCoord*)
  - Breedte? (*breedte*)
  - Hoogte? (*hoogte*)

  Gegenereerde code:

```
pen.drawOval(xCoord, yCoord, breedte, hoogte);
```

- **Teken een gevulde ovaal**
  Opties die gegeven kunnen worden:
  - x-coördinaat linksbovenhoek? (*xCoord*)
  - y-coördinaat linksbovenhoek? (*yCoord*)
  - Breedte? (*breedte*)
  - Hoogte? (*hoogte*)

  Gegenereerde code:

```
pen.fillOval (xCoord, yCoord, breedte, hoogte);
```

- **Verander de tekenkleur**
  Opties die gegeven kunnen worden:
  - Geef de nieuwe kleur. Kies uit: black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, yellow. (*kleur*)

  Gegenereerde code:

```
pen.setColor(kleur);
```

GUI

In deze tab staan elementen van de AWT.

- **Maak een knop (Button)**
  Opties die gegeven kunnen worden:
  - Wat is de naam van de knop? (*knopNaam*)
  - Wat is het opschrift van de knop? (*opschrift*)
  - Welke methode moet worden uitgevoerd, als er op de knop wordt geklikt? (*actionMethode*)

  Gegenereerde code:
  - In `VariabeleDeclaratieVanGUI`:

```
Button knopNaam;
```

  - In `InstructiesInit`:

```
knopNaam = new Button(opschrift);
add(knopNaam);
```

  - In `InstructiesAction`:

```
if (e.target == knopNaam)
{
    this.actionMethode();
}
```

- **Maak een invoervak (TextField)**
  Opties die gegeven kunnen worden:
  - Wat is de naam van het vakje? (*vakNaam*)
  - Wat is het de breedte van het vakje (uitgedrukt in het aantal letters, bijv. 20)? (*breedte*)

  Gegenereerde code:
  - In `VariabeleDeclaratieVanGUI`:

```
TextField vakNaam;
```

  - In `InstructiesInit`:

```
vakNaam = new TextField(breedte);
add(vakNaam);
```

- **Maak een vast tekstvak (Label)**
  Opties die gegeven kunnen worden:
  - Wat is de naam van het tekstvak? (*labelNaam*)
  - Welke tekst moet erin? (*tekst*)

  Gegenereerde code:
  - In `VariabeleDeclaratieVanGUI`:

```
Label labelNaam;
```

  - In `InstructiesInit`:

```
labelNaam = new Label(opschrift);
add(labelNaam);
```

- **Stel de positie van een element in**
  Opties die gegeven kunnen worden:
  - Van welk element? (*elementNaam*)
  - x-coördinaat linksbovenhoek? (*xCoord*)
  - y-coördinaat linksbovenhoek? (*yCoord*)
  - Breedte? (*breedte*)
  - Hoogte? (*hoogte*)

  Gegenereerde code:

```
elementNaam.setBounds(xCoord, yCoord, breedte, hoogte);
```

- **De tekst op een knop aanpassen**
  Opties die gegeven kunnen worden:
  - Van welke knop? (*knopNaam*)
  - Wat is de nieuwe tekst? (*nieuweTekst*)

  Gegenereerde code:

```
knopNaam.setLabel(nieuweTekst);
```

- **De tekst in een invoervak aanpassen**
  Opties die gegeven kunnen worden:
  - Van welk invoervak? (*vakNaam*)
  - Wat is de nieuwe tekst? (*nieuweTekst*)

  Gegenereerde code:

```
vakNaam.setText(nieuweTekst);
```

- **De tekst in een vast tekstvak aanpassen**
  Opties die gegeven kunnen worden:
  - Van welk tekstvak? (*labelNaam*)
  - Wat is de nieuwe tekst? (*nieuweTekst*)

  Gegenereerde code:

```
labelNaam.setText(nieuweTekst);
```

JavaLogo

De tab JavaLogo is nog niet aanwezig in de eerste versie van het programma. Misschien wordt dit later tijdens dit project ingebouwd, of bij een latere versie.

Vanaf hier ga ik verder met een “tussenproject”: de PSD Creator. De bespreking van het hoofdprogramma gaat verder op pagina 35.

De PSD Creator

Ik heb besloten om, vóór verder te gaan met stap 3 van het hoofdprogramma, een klein programmaatje te programmeren, genaamd PSD Creator. Ook voor dit programmaatje volg ik de zes stappen.

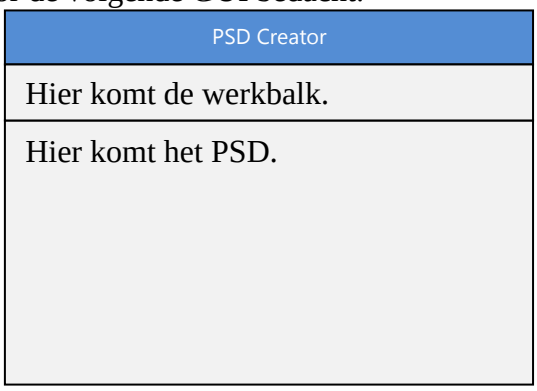
Stap 1: Probleem definiëren

Het programma moet twee dingen kunnen:

- een PSD, bestaande uit de drie soorten blokken (opdrachtblok, herhalingsblok, keuzeblok) kunnen tekenen
- dit PSD in het programma kunnen bewerken

Het programma is bedoeld om als subprogramma te dienen voor de Applet Creator (zodat de code weer gebruikt kan worden voor de Applet Creator) maar het zou ook als zelfstandig programma bruikbaar moeten zijn. Ik ga daaraan echter niet al te veel aandacht besteden.

Ik heb voor de PSD Creator de volgende GUI bedacht:



Figuur 7 – Ontwerp voor de GUI van de PSD Creator

De werkbalk krijgt de volgende knoppen:

- **nieuw opdrachtblok** – voeg aan het einde van het PSD een nieuw opdrachtblok toe. Als hierop wordt geklikt, dan moet de gebruiker een opdracht ingeven in een standaard Swing-dialoogje.
- **nieuw herhalingsblok** – voeg aan het einde van het PSD een nieuw, leeg herhalingsblok toe. Als hierop wordt geklikt, dan moet de gebruiker een herhalingsvoorwaarde ingeven in een standaard Swing-dialoogje.
- **nieuw keuzeblok** – voeg aan het einde van het PSD een nieuw, leeg keuzeblok toe. Als hierop wordt geklikt, dan moet de gebruiker een vraag ingeven in een standaard Swing-dialoogje.
- **blok omhoog** – werkt op dezelfde manier als het knopje *omhoog* in de Applet Creator.
- **blok omlaag** – werkt op dezelfde manier als het knopje *omlaag* in de Applet Creator.
- **blok verwijderen** – verwijdert het geselecteerde blok.

Het PSD wordt op de rest van het scherm getekend. Er is geen menubalk. Verder is het niet mogelijk om:

- het PSD helemaal te wissen
- het PSD op te slaan of te openen
- het PSD te exporteren als plaatje, dit moet als een screenshot

Stap 2: Probleem analyseren

Ik moet een manier bedenken waarop de blokken eenvoudig kunnen worden opgeslagen en getekend. Deze manier moet aan de volgende eisen voldoen.

- De opslag moet zó gebeuren dat het tekenen makkelijk gaat. Dit kan bijvoorbeeld door alle blokken, die door een herhalings- of keuzeblok ingesloten worden, bij dit herhalings- of keuzeblok op te slaan; dus ieder herhalings- en keuzeblok heeft dan één of twee Vectoren waarin de blokken komen. Het voordeel hiervan is dat het herhalings- of keuzeblok als klasse gemakkelijk kan bepalen welke blokken erin moeten worden getekend. Een nadeel is dat het een enorm probleem wordt om één individueel blok te bereiken.
- De opslag moet zó gebeuren dat het verschuiven makkelijk gaat. Dit kan bijvoorbeeld door alle blokken centraal op te slaan in één klasse, in één Vector. Verschuiven kan dan makkelijker omdat alles centraal opgeslagen ligt. Een probleem hiervan is dat het moeilijk is om aan te geven welke blokken bij welke andere blokken horen.

Vectoren

Een Vector is een soort array. Net als een array bevat hij een rij met variabelen of objecten. Een nadeel van een array is dat deze niet langer of korter gemaakt kan worden tijdens de uitvoering. Met een Vector kan dat wel.
Vector is gedefinieerd in de package `java.util.Vector`.

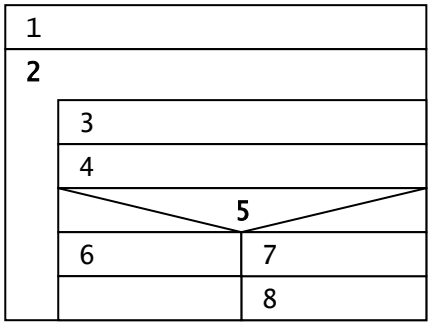
De hierboven genoemde eisen zijn tegenstrijdig. De ene wil dat alle blokken bij elkaar worden opgeslagen in één Vector, de andere wil dat ieder blok een eigen Vector krijgt! Ik heb hiervoor de volgende oplossing gevonden.

- De hoofdklasse van het programma maakt één Vector aan waarin het PSD wordt opgeslagen.
- Deze Vector kan dan de volgende elementen bevatten.
  - *GewoonBlok* – een gewoon opdrachtblok
  - *HerhalingsBlok* – een gewoon herhalingsblok
  - *EindHerhaling* – afsluiting van het herhalingsblok, equivalent met de accolade sluiten in Java
  - *KeuzeBlok* – een gewoon keuzeblok met *ja* en *nee*
  - *JaNeeWissel* – afsluiting van de ja-kolom en begin van de nee-kolom, equivalent met `} else {` in Java
  - *EindKeuze* – afsluiting van het keuzeblok, equivalent met de accolade sluiten in Java
- Tekenen gebeurt in PSD. Dit kan, omdat goed wordt aangegeven wat waarbij hoort.
- Verschuiven gaat makkelijk, want alles staat in één Vector.

Tekenen

Stel: in de Vector staan achtereenvolgens de volgende elementen:

- Gewoon blok (1)
- Herhalingsblok (2)
- Gewoon blok (3)
- Gewoon blok (4)
- Keuzeblok (5)
- Gewoon blok (6)
- Ja-nee-wissel
- Gewoon blok (7)



Figuur 8 – Voorbeeld van een PSD zoals dat getekend moet worden door het tekenalgoritme

- Gewoon blok (8)
- Eind Keuze
- Eind Herhaling

Dit zou dan getekend moeten worden zoals in *Figuur 8*. Nu moet ik een tekenalgoritme gaan bedenken dat het tekenen uitvoert.
Het algoritme gebruikt de variabele *hoogte*. Deze is *public* en kan door alle methoden worden gewijzigd.

void hoofdTekenMethode:

- Stel *hoogte* in op 0.
- Voer **tekenPSD(0, *schermbreedte*, 0)** uit. Met het resultaat hoeft niets te worden gedaan.

int tekenPSD(xLinks, xRechts, startPositie):
(tekenPSD geeft de plaats in de Vector terug, waar gestopt is met tekenen.)

- Doe het volgende voor ieder element in de Vector vanaf positie startPositie. Gebruik als teller de variabele *positie*.
  1. Als het element een Gewoon blok is dan
     a. Teken een gewoon blok met *xLinks*, *xRechts*, *hoogte*, *hoogte* + 20.
     b. Vermeerder hoogte met 20.
  2. Als het element een Keuzeblok is dan
     a. Teken een keuzeblok met *xLinks*, *xRechts*, *hoogte*, *hoogte* + 30.
     b. Vermeerder hoogte met 30.
     c. Sla hoogte op in hoogteOnderKeuzeblok.
     d. Voer **tekenPSD(*xLinks*, $\frac{xLinks+xRechts}{2}$, *positie*+1)** uit. Zet de resultaatwaarde op *positie*.
     e. Sla hoogte op in hoogteOnderJa.
     f. Stel hoogte in op hoogteOnderKeuzeblok.
     g. Voer **tekenPSD($\frac{xLinks+xRechts}{2}$, *xRechts*, *positie*)** uit. Zet de resultaatwaarde-1 op *positie*.
     h. Sla *hoogte* op in *hoogteOnderNee*.
     i. Als *hoogteOnderJa* > *hoogteOnderNee*
        i. Teken een rechthoek met $\frac{xLinks+xRechts}{2}$, *hoogteOnderNee*, *xRechts* - $\frac{xLinks+xRechts}{2}$, *hoogteOnderJa – hoogteOnderNee*
        ii. Stel *hoogte* in op *hoogteOnderJa*.
     j. Als hoogteOnderNee > hoogteOnderJa
        i. Teken een rechthoek met *xLinks*, *hoogteOnderJa*, $\frac{xRechts-xLinks}{2}$, *hoogteOnderNee* - *hoogteOnderJa*
        ii. Stel *hoogte* in op *hoogteOnderNee*.
  3. Als het element een Herhalingsblok is dan
     a. Teken de bovenkant van het blok (met *xLinks*, *xRechts*, *hoogte*, *hoogte* + 20). Teken van de rand alleen de bovenkant en de rechter zijkant.
     b. Sla hoogte op in hoogteBijBeginTekenen.
     c. Vermeerder de hoogte met 20.
     d. Voer **tekenPSD(*xLinks* + *20*, *xRechts*, *positie* + 1)** uit. Zet de resultaatwaarde-1 op *positie*.

e. Teken de resterende randen van het blok erbij, dat zijn de linkerkant en de onderkant.
f. Vul de linker zijkant van het blok (die langs de sub-blokken loopt) met wit op.

4. Als het element een Eind Keuze, Eind Herhaling of Ja-nee-wissel is, dan zorg dat de for-loop stopt.

- Geef terug (*return*) de positie.

Verschuiven

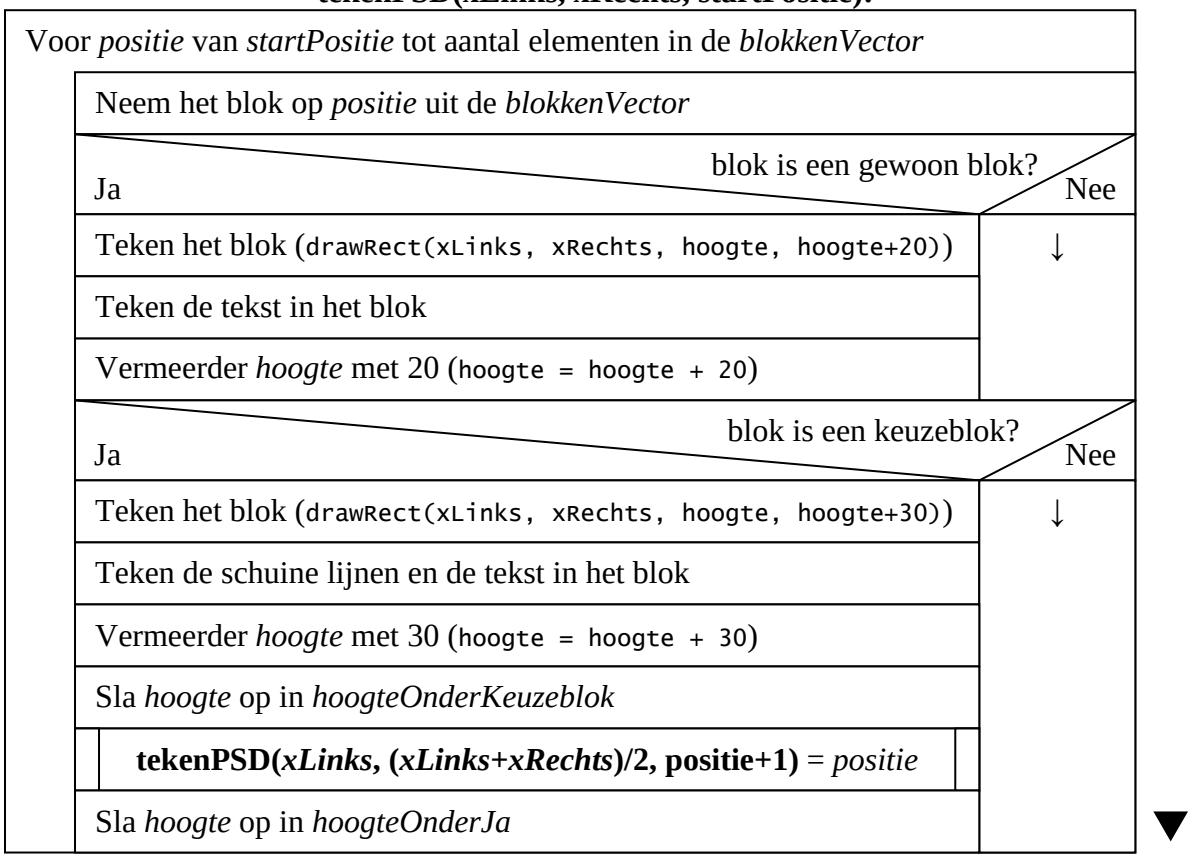
Het verschuiven van blokken kan betrekkelijk eenvoudig als het gaat om gewone blokken die verschoven worden. De Vector, waarin de blokken staan, heeft als voordeel dat er (tenminste, als er een gewoon blok verschoven wordt) alleen maar twee elementen in de Vector hoeven worden omgedraaid bij de verschuiving: als blok *n* in de Vector omhooggeschoven wordt, dan worden element *n-1* en *n* omgewisseld; als blok *n* in de Vector omlaaggeschoven wordt, dan worden element *n* en *n+1* omgewisseld.

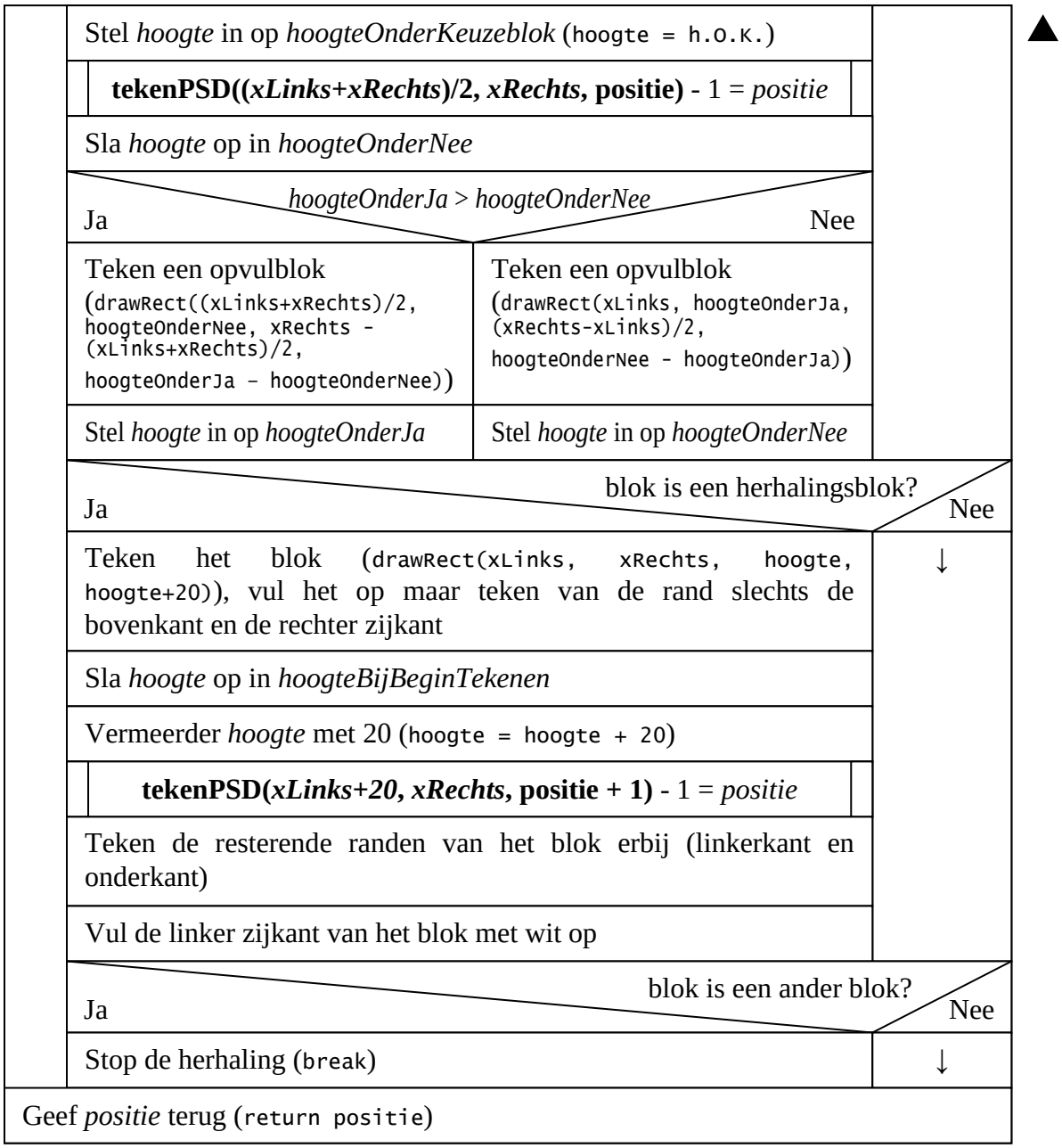
Dit geldt voor alle speciale gevallen die gegeven zijn in de paragraaf Blokken toevoegen en verplaatsen van stap 1 van de Applet Creator (pag. 7 in deze documentatie).
Er ontstaat echter een probleem als er een herhalingsblok of een keuzeblok wordt verplaatst. De bedoeling is namelijk dat dan het volledige blok, inclusief inhoud, wordt verplaatst. Dit werkt dus op dezelfde manier als bij het verplaatsen bij een gewoon blok, alleen moeten dan meerdere blokken tegelijkertijd verplaatst worden (keuze- of herhalingsblok met de inhoud erbij).

Stap 3: Oplossing schematiseren

Tekenen

tekenPSD(xLlinks, xRechts, startPositie):

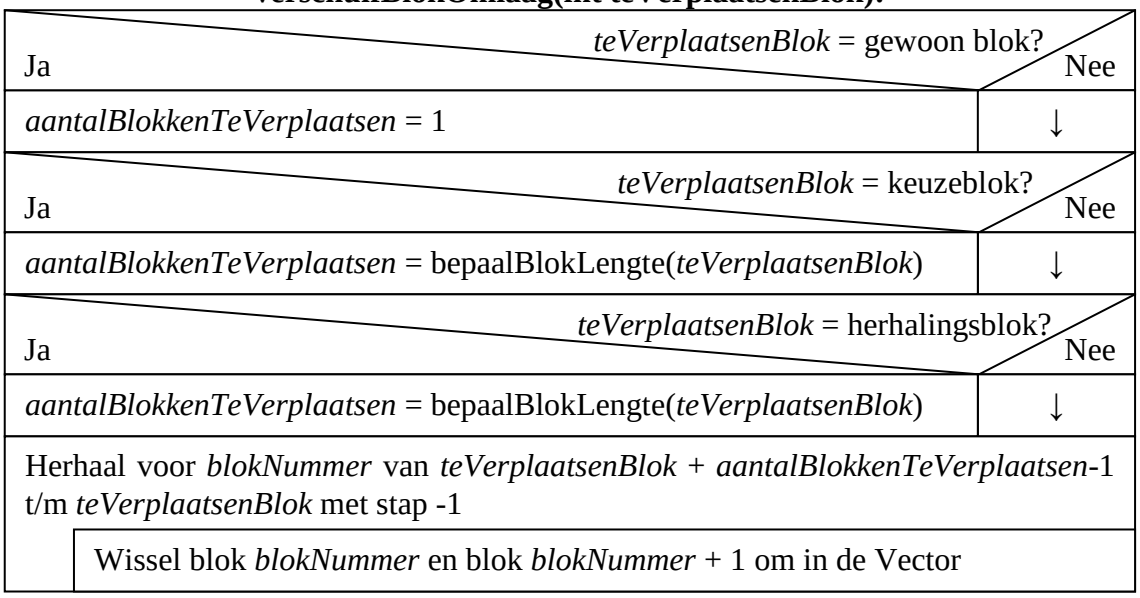



Stel *hoogte* in op *hoogteOnderKeuzeblok* (hoogte = h.O.K.)
tekenPSD((*xLlinks*+*xRechts*)/2, *xRechts*, positie) - 1 = *positie*
Sla *hoogte* op in *hoogteOnderNee*
hoogteOnderJa > *hoogteOnderNee*
Ja
Nee
Teken een opvulblok (drawRect((xLlinks+xRechts)/2, hoogteOnderNee, xRechts - (xLinks+xRechts)/2, hoogteOnderJa – hoogteOnderNee))
Teken een opvulblok (drawRect(xLinks, hoogteOnderJa, (xRechts-xLinks)/2, hoogteOnderNee - hoogteOnderJa))
Stel *hoogte* in op *hoogteOnderJa*
Stel *hoogte* in op *hoogteOnderNee*
blok is een herhalingsblok?
Ja
Nee
Teken het blok (drawRect(xLinks, xRechts, hoogte, hoogte+20)), vul het op maar teken van de rand slechts de bovenkant en de rechter zijkant
Sla *hoogte* op in *hoogteBijBeginTekenen*
Vermeerder *hoogte* met 20 (hoogte = hoogte + 20)
tekenPSD(*xLinks*+*20*, *xRechts*, positie + 1) - 1 = *positie*
Teken de resterende randen van het blok erbij (linkerkant en onderkant)
Vul de linker zijkant van het blok met wit op
blok is een ander blok?
Ja
Nee
Stop de herhaling (break)
Geef *positie* terug (return positie)

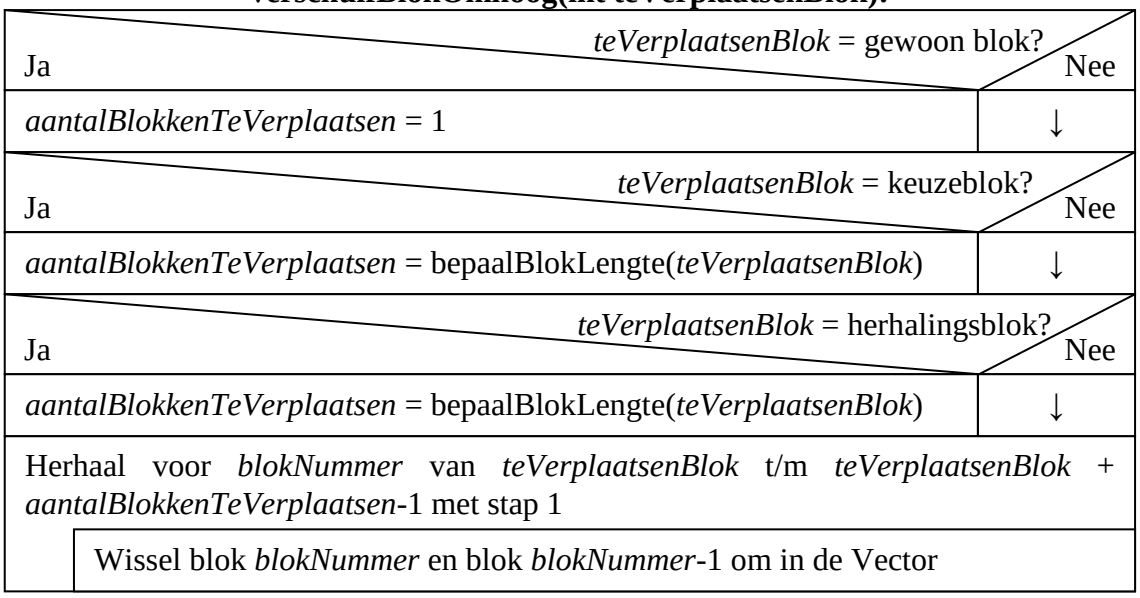
Verschuiven

aantalBlokkenTeVerplaatsen geeft weer hoeveel blokken er moeten worden verplaatst. Het algoritme bepaalt dit eerst; hierna wordt het blok verschoven.

verschuifBlokOmlaag(int teVerplaatsenBlok):



verschuifBlokOmhoog(int teVerplaatsenBlok):



Dit algoritme heeft de methode bepaalBlokLengte nodig:

bepaalBlokLengte(int blokIndex)

teller = 0
nesting = 0
Herhaal voor alle blokken in de Vector vanaf *blokIndex*
Als blok = keuzeblok of herhalingsblok, dan *nesting* 1 verhogen
Als blok = eind keuze of eind herhaling, dan *nesting* 1 verlagen
teller met 1 verhogen
Als *nesting* is 0, dan stop de herhalingslus
Geef *teller* terug

Selecteren

Om een blok te selecteren, bijvoorbeeld om het te verplaatsen of het te verwijderen, moet er eerst onderzocht worden waar (x- en y-coördinaat) er is geklikt. Hierna wordt dezelfde recursieve methode toegepast als bij het tekenen, alleen wordt er hier gekeken per blok of de x- en y-coördinaten van de muisklik binnen dit blok vallen, in plaats van het blok te tekenen. Als gevonden wordt waarop geklikt is, wordt de selectie opnieuw ingesteld en wordt het tekenvel opnieuw getekend.

Stap 4: Programma-instructies coderen

Nu ga ik de PSD Creator maken in Java.

Klassenindeling

Ik zal de volgende klassenindeling gebruiken voor het programma:

- **PSDCreator**, het hoofdprogramma. Deze klasse maakt een instantie van PSD.
- **PSD**, dit is het JPanel waarop het PSD zal worden getekend. Deze klasse zorgt voor het bewaren en het tekenen van het PSD, voor het bepalen van het blok waarop geklikt is (om het te selecteren), voor het toevoegen van blokken en voor het schuiven van blokken door het PSD. Deze klasse heeft een Vector met daarin alle Blokken (zie hieronder).
- **Blok**, een klasse die alleen twee Strings verzamelt, namelijk één met het label van het blok (dat is de opdracht, de vraag of de herhalingsvoorwaarde die op het blok geschreven wordt) en één met het soort blok.
- **Werkbalk**, dit is de werkbalk van het hoofdvenster.
- **ActieAfhandelaar**, de code die ervoor zorgt dat er iets gebeurt als er op een knop in de werkbalk geklikt wordt.

Klasse PSDCreator

Deze klasse is het hoofdprogramma. Achtereenvolgens worden er de volgende taken uitgevoerd:

- Het JFrame wordt gemaakt.
- Er wordt een contentpane gemaakt, waaraan een PSD en een Werkbalk worden toegevoegd.

- Het venster wordt op de goede grootte gemaakt, en het wordt op het scherm weergegeven.

Klasse PSD

Deze klasse doet heel veel. Hij zorgt voor:

- het tekenen van het PSD
- het verplaatsen van blokken
- het afhandelen van een muisklik op het PSD
- het verwijderen van blokken

In deze klasse wordt een Vector aangemaakt, *blokkenVector*, waarin alle blokken achter elkaar worden opgeslagen. Dit gebeurt volgens het systeem zoals beschreven op pagina 20.

De methoden van deze klasse zijn:

- **paintComponent**: deze wordt automatisch aangeroepen bij het verversen van het PSD. Hierin wordt onder meer tekenPSD aangeroepen.
- **tekenPSD**: tekent het PSD. Deze methode roept zichzelf nog eens aan om de blokken binnenin keuze- en herhalingsblokken te tekenen. De methode werkt volgens het PSD *tekenPSD*.
- **voegToeOpdrachtblok**, **voegToeHerhalingsblok**, **voegToeKeuzeblok**: voegt een blok van het desbetreffende type toe aan het PSD. Alle vragen een naam die op het blok komt te staan; deze kan dan door klasse ActieAfhandelaar gevraagd worden in een dialoogvenster.
- **verschuifBlokOmlaag**, **verschuifBlokOmhoog**: verschuift het blok, waarvan de index als parameter wordt meegegeven, omlaag of omhoog. Als het blok andere blokken bevat, schuiven die automatisch mee. Bij deze methoden hoort de methode **bepaalBlokLengte** die bij een herhalings- of keuzeblok bepaalt hoeveel blokken er verschoven moeten worden en de methode **blokkenInVectorOmwisselen** die twee opgegeven blokken omdraait in de Vector.
- **zoekGekliktBlok**: deze werkt het geselecteerde blok bij naar aanleiding van een muisklik. Eigenlijk werkt deze methode hetzelfde als tekenPSD, echter er worden geen blokken getekend; in plaats daarvan wordt bij ieder blok gekeken of de muisklik binnen dat blok valt.
- **verwijderBlok**: verwijdert het opgegeven blok en alle sub-blokken in dat blok.
- De methoden die komen van MouseListener: **mousePressed**, **mouseReleased** enz. Deze zijn allemaal leeg, behalve **mouseClicked**; deze methode roept namelijk de methode zoekGekliktBlok aan om de selectie te veranderen.

Verder is er nog een simpele constructor zonder parameters die de grootte instelt en een MouseListener toevoegt aan de klasse.

Hieronder wordt van enkele methoden de werking nog nader toegelicht.

Methode *bepaalBlokLengte*

Deze methode retourneert de lengte van het blok, waarvan de index in de Vector als parameter meegegeven wordt.
De lengte van een bepaald blok X wordt hier gedefinieerd als *het aantal blokken in de Vector dat gebruikt moet worden, om het blok X (inclusief blokken die binnenin blok X zitten) te kunnen tekenen*.
Als voorbeeld nemen we hier de volgende blokkenVector:

- Gewoon
- Herhaling
- Gewoon
- Keuze
- Gewoon
- Ja-nee
- Gewoon
- Gewoon
- Eind Keuze
- Eind Herhaling
- Gewoon
- Gewoon

De lengte van het blok Herhaling op index 1 is 9; namelijk de hierna vetgedrukte blokken horen bij Herhaling:

- Gewoon
- **Herhaling**
- **Gewoon**
- **Keuze**
- **Gewoon**
- **Ja-nee**
- **Gewoon**
- **Gewoon**
- **Eind Keuze**
- **Eind Herhaling**
- Gewoon
- Gewoon

De lengte van een blok is belangrijk, omdat deze nodig is voor het verplaatsen en verwijderen van blokken.

- Willen we Herhaling namelijk verwijderen, dan moeten alle vetgedrukte blokken verwijderd worden. Dit kan eenvoudig met een for-loop, als we tenminste weten hoeveel blokken er verwijderd moeten worden.
- Willen we Herhaling naar onderen verplaatsen, dan moeten alle vetgedrukte blokken verplaatst worden. Dit kan ook weer met een for-loop, als we weten hoeveel blokken er verplaatst moeten worden.

Methoden *verschuifBlokOmlaag* en *verschuifBlokOmhoog*

Deze methoden werken met een for-loop. De werking daarvan wordt hieronder toegelicht. Ik gebruik hier *verschuifBlokOmhoog*, maar hetzelfde geldt ook voor de omlaag-variant.
De for-loop die bij *verschuifBlokOmhoog* gebruikt wordt, is:

```
        for (int blokNummer = teVerplaatsenBlok; blokNummer <
              teVerplaatsenBlok + aantalBlokkenTeVerplaatsen; blokNummer++)
        {
            blokkenInVectorOmwisselen(blokNummer, blokNummer-1);
        }
```

teVerplaatsenBlok is hierbij de index in de Vector van het blok dat verplaatst moet worden; *aantalBlokkenTeVerplaatsen* is de lengte van het *teVerplaatsenBlok*.

Aan de hand van een voorbeeld laat ik hier zien hoe de for-loop werkt. We nemen als blokkenVector:

- Gewoon
- Gewoon
- **Herhaling**
- **Gewoon**
- **Eind Herhaling**
- Gewoon

We gaan de methode nu uitvoeren met *teVerplaatsenBlok* = 2 (dus Herhaling). *aantalBlokkenTeVerplaatsen* is 3, want er horen 3 blokken bij dit Herhalingsblok (de vetgedrukte blokken).
Als we nu dit blok omhoog gaan verplaatsen, wisselen we eerst blok 1 en blok 2 om. Het resultaat is:

- Gewoon
- **Herhaling**
- Gewoon
- **Gewoon**
- **Eind Herhaling**
- Gewoon

Nu wisselen we blok 2 en blok 3 om:

- Gewoon
- **Herhaling**
- **Gewoon**
- Gewoon
- **Eind Herhaling**
- Gewoon

En als laatste wisselen we blok 3 en blok 4 om:

- Gewoon
- **Herhaling**
- **Gewoon**
- **Eind Herhaling**
- Gewoon
- Gewoon

Het blok is nu opgeschoven.
De for-loop voert precies het hierboven beschreven algoritme uit. Er loopt namelijk een teller van *teVerplaatsenBlok* tot (dus niet tot en met) *teVerplaatsenBlok* + *aantalBlokkenTeVerplaatsen*, dus hier van 2 tot 5. De for-loop wisselt in iedere cyclus de blokken op *teller*-1 en *teller* om, dus hier worden achtereenvolgens 1 en 2 omgewisseld, daarna 2 en 3 en daarna 3 en 4. Dit is dus precies het hierboven beschreven algoritme.

De methode *verschuifBlokOmlaag* werkt hetzelfde, met het verschil dat hierbij de for-loop achteruitloopt omdat hier onderaan het blok begonnen moet worden.

Methode *zoekGekliktBlok*

Deze methode gebruikt bij ieder blok dat hij in de Vector tegenkomt, een if-statement dat controleert of de muisklik binnen het blok valt. Als dat het geval is, dan wordt *selectie* ingesteld op dit blok.

Klasse ActieAfhandelaar

Deze klasse zorgt voor het afhandelen van acties die plaatsvinden op de werkbalk. Als er wordt geklikt op een knop op de werkbalk, dan onderzoekt deze klasse welke actie daarbij hoort en voert deze uit. Het uitvoeren gebeurt door simpele aanroepen van methoden van PSD.

Klasse Blok

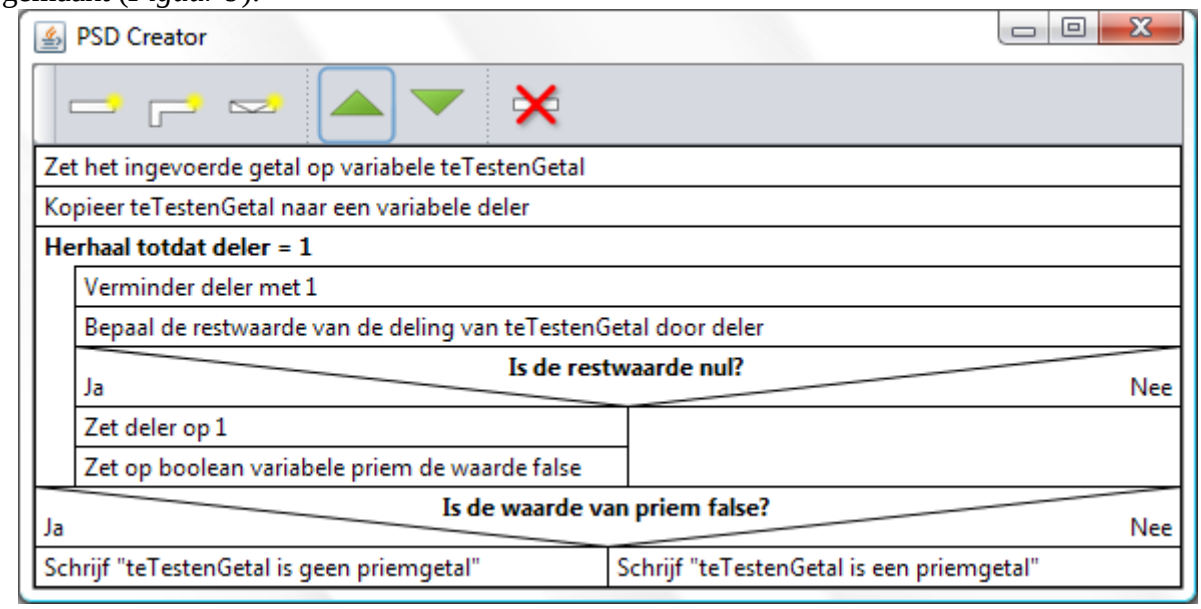
Een Blok is heel simpel: hij hoeft alleen maar twee Strings te hebben. Deze zijn *public* zodat ze gewijzigd en opgevraagd kunnen worden.
De twee Strings zijn:

- *typeBlok* die het soort blok weergeeft, bijvoorbeeld Gewoon of Herhaling;
- *label* die weergeeft wat er op het blok komt te staan, bijvoorbeeld de opdracht of de vraag.

Klasse Werkbalk

Deze klasse is afgeleid van JToolBar en bevat, naast de constructor die de knoppen in de werkbalk zet, ook een methode knopMaken die wordt gebruikt door de constructor. Deze methode maakt een JButton en geeft deze terug.

Nu alle klassen geprogrammeerd zijn, heb ik een schermafdruk van het huidige programma gemaakt (*Figuur 9*).



Figuur 9 – Een schermafdruk van de huidige versie van PSD Creator. In het programma staat een PSD om te bepalen of een getal priem is. (Dit algoritme komt uit het boekje Java: my cup of tea.*)*

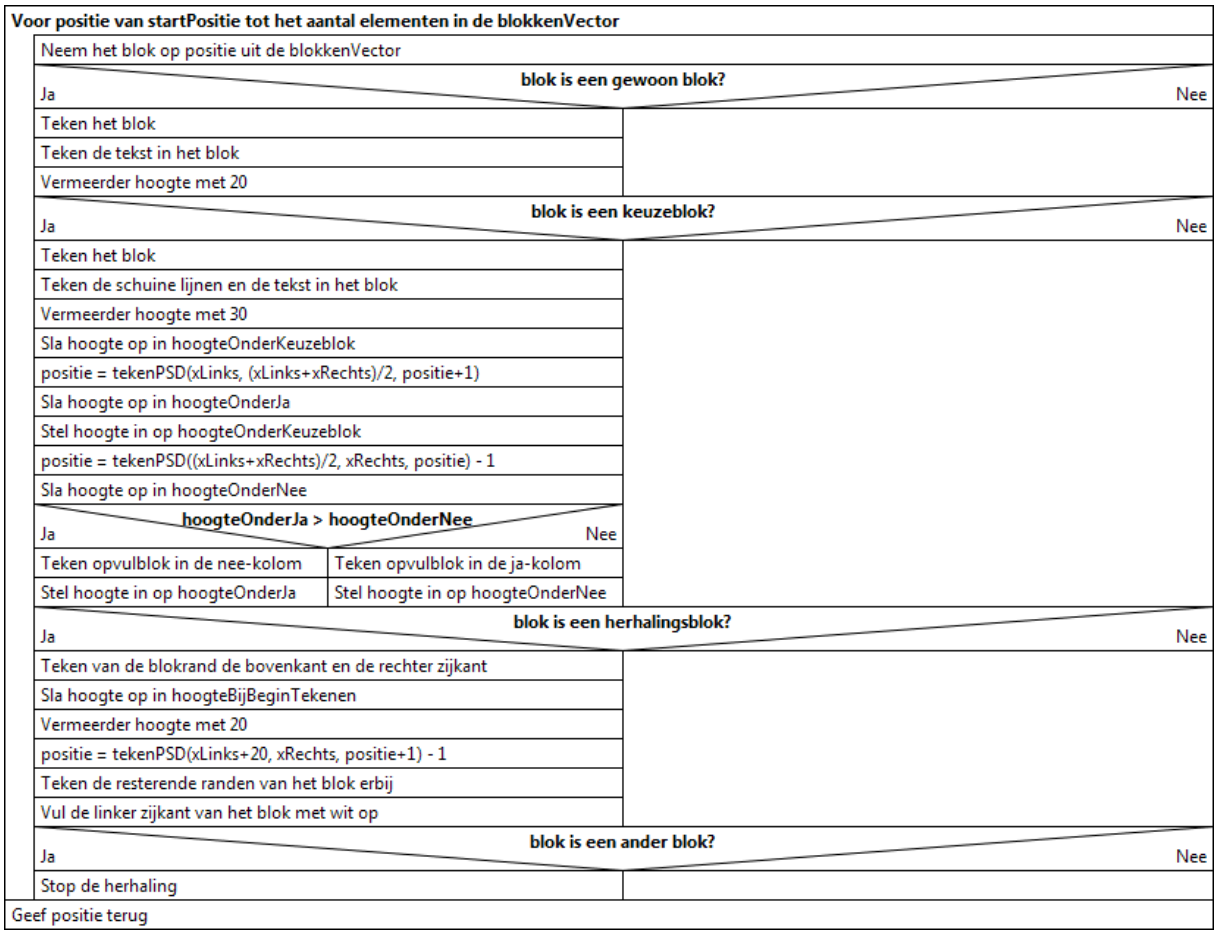
Stap 5: Testen

De test bestaat uit twee delen:

- Het eerste deel bestaat uit het maken van enkele ingewikkelde PSD's, om te kijken of dit goed gaat.
- Het tweede deel bestaat uit het testen van specifieke probleemgevallen, zoals het verschuiven van een keuze- of herhalingsblok of het verwijderen daarvan.

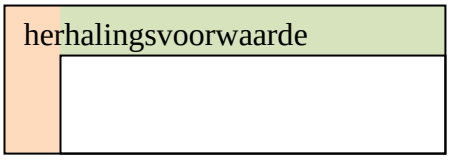
Ingewikkelde PSD's testen

Het eerste PSD dat ik gemaakt heb, is het PSD voor het tekenen van het PSD voor dit programma. Dit PSD staat in deze documentatie op pagina 22. In *Figuur 10* staat het resultaat van deze eerste test.



Figuur 10 – Het resultaat van de eerste test

Ik heb één fout gevonden in het programma tijdens het maken van dit PSD. Als je namelijk klikt op de linker zijkant van een herhalingsblok, dan wordt het blok niet geselecteerd, terwijl dat wel de bedoeling is. Zie *Figuur 11*. Als er op het groene gedeelte van het herhalingsblok geklikt wordt, dan wordt dit geselecteerd; als echter op het oranje gedeelte geklikt wordt, dan blijft het blok niet geselecteerd.



Figuur 11 – Als er op het groene gedeelte wordt geklikt, dan wordt het blok geselecteerd.

Hieronder staat de code van de controle op de linker zijkant van een herhalingsblok:

```
    if (klikX > xLinks && klikX < (xLinks+20) && klikY >
                                    hoogteBijBeginTekenen && klikY < hoogte)
    {
        selectie = positie;
    }
```

Op zichzelf genomen klopt deze code wel, maar de regel ervoor is:

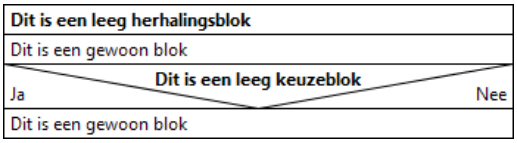
```
    positie = zoekGekliktBlok(xLinks + 20, xRechts, positie + 1, klikX,
                                                                 klikY) - 1;
```

De positie wordt dus aangepast, en het blok dat geselecteerd wordt is dan Eind Herhaling! De oplossing van het probleem is dus om positie tijdelijk op te slaan:

```
int positieVanHerhalingsblok = positie;
positie = zoekGekliktBlok(xLinks + 20, xRechts, positie + 1, klikX,
                                                          klikY) - 1;

if (klikX > xLinks && klikX < (xLinks+20) && klikY >
                                hoogteBijBeginTekenen && klikY < hoogte)
{
    selectie = positieVanHerhalingsblok;
}
```

Verder zijn er nog enkele verbeterpunten:

- Als één kant van een keuzeblok leeg is, dan zou daarin een pijltje omlaag moeten staan, zoals dat hoort.
- Een leeg keuzeblok en vooral een leeg herhalingsblok zijn niet duidelijk. Zie *Figuur 12*. Het probleem is dat het voor de gebruiker niet duidelijk is dat er blokken in deze blokken gezet kunnen worden. Dit probleem is vooral duidelijk bij het herhalingsblok: het herhalingsblok ziet er (behalve de vette tekst) net zo uit als een gewoon blok. Eigenlijk zou een leeg herhalingsblok getekend moeten worden alsof er al een gewoon blok (zonder tekst) inzit.



Figuur 12 – Lege keuze- en herhalingsblokken worden eigenlijk niet goed getekend

Deze verbeterpunten ga ik nu ook nog inbouwen.
Als eerste: hoe bepaal ik of een herhalings- of keuzeblok "leeg" is? Dat kan door te controleren of de hoogte na het tekenen van het "hoofdblok" gelijk is aan de hoogte na het tekenen van de sub-blokken.

Een leeg herhalings- of keuzeblok is niet duidelijk

Hier moet ervoor gezorgd worden dat een leeg blok beter getekend wordt. Als eerste het herhalingsblok: het aanpassen hiervan gebeurt met het volgende stukje code in tekenPSD:

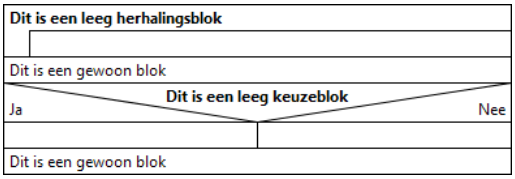
```
if (hoogteBijBeginTekenen + 20 == hoogte)
{
    // TEKEN LEEG BLOK
    hoogte = hoogte + 20;
}
```

Dit controleert of het herhalingsblok leeg is. Als dit het geval is, dan wordt eerst een leeg blok getekend en daarna wordt de hoogte vermeerderd met 20, zodat het blok op de goede manier in het herhalingsblok komt te staan. Uiteraard moet ook de code in zoekGekliktBlok aangepast worden aan de nieuwe tekenmethode.
Het keuzeblok is iets ingewikkelder: hierbij moet worden gecontroleerd of beide kanten van het blok leeg zijn; als dit het geval is, dan moet er aan allebei de kanten een leeg blok geplaatst worden. De voorwaarde hierbij is:

```
if (hoogteOnderJa == hoogteOnderKeuzeblok && hoogteOnderNee ==
                                                    hoogteOnderKeuzeblok)
```

Ook hierbij moet zowel tekenPSD als zoekGekliktBlok aangepast worden.

Nadat dit was ingebouwd, heb ik nog een keer het PSD van *Figuur 12* gemaakt. Dit ziet er nu duidelijker uit, zie *Figuur 13*.

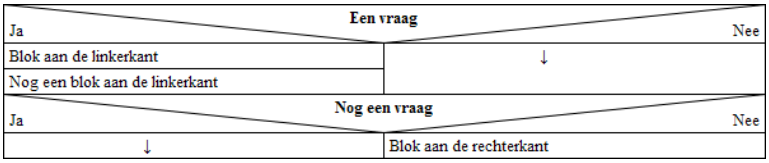


Figuur 13 – Lege blokken worden nu beter getekend

Een pijltje in het keuzeblok bij een lege kolom

Als alleen de ja- óf de nee-kolom van het keuzeblok leeg is, dan moet er in deze kolom gecentreerd een pijltje naar beneden geplaatst worden. Ik heb besloten dat er geen pijltjes geplaatst moeten worden als beide kolommen leeg zijn. Aangezien ik al een if-blok heb waarin gecontroleerd wordt of beide kolommen leeg zijn, kan ik met een else-if-constructie controleren of de ja-kolom leeg is, en daarna ook de nee-kolom.
De structuur wordt dan dus:

```
if (hoogteOnderJa == hoogteOnderKeuzeblok && hoogteOnderNee
  == hoogteOnderKeuzeblok)   // allebei de kanten leeg?
{
    // LEGE BLOKKEN TEKENEN (zie vorige pagina)
}
else if (hoogteOnderJa == hoogteOnderKeuzeblok)
                                  // alleen linkerkant leeg?
{
    // TEKEN PIJL AAN DE LINKERKANT
}
else if (hoogteOnderNee == hoogteOnderKeuzeblok)
                                  // alleen rechterkant leeg?
{
    // TEKEN PIJL AAN DE RECHTERKANT
}
```

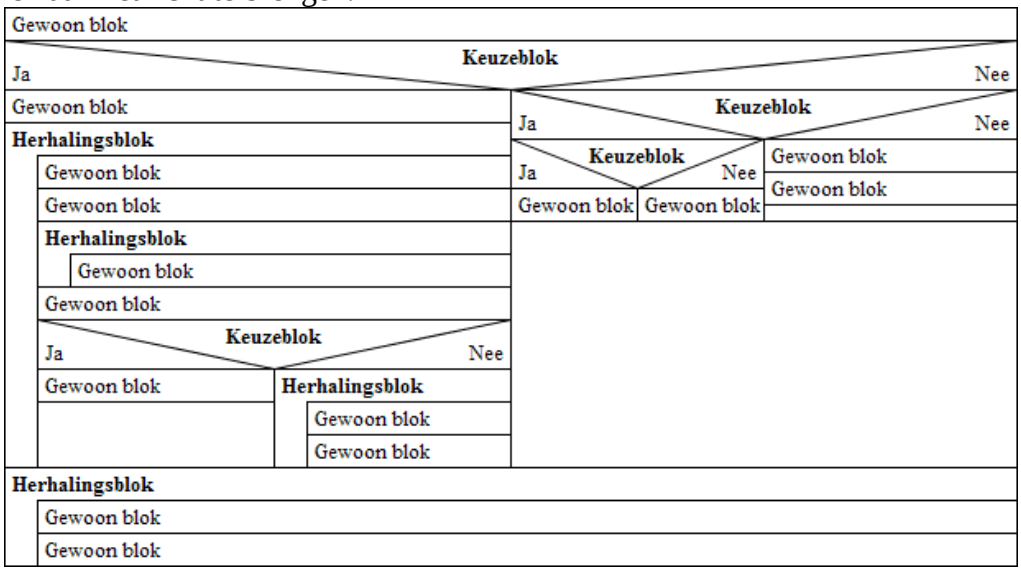


Figuur 14 – Voorbeeld van pijltjes in keuzeblokken waarvan één kant leeg is

Het centreren van de pijlen doe ik op dezelfde manier waarmee ik ook de tekst van een keuzeblok laat tekenen. Ik moest wel een ander lettertype kiezen voor het PSD, omdat Segoe UI noch Tahoma het Unicode-teken 2193 voor een pijltje ondersteunde. Nu gebruik ik Times New Roman. Een voorbeeld van pijltjes in keuzeblokken staat in *Figuur 14*.

Specifieke probleemgevallen testen

Ik ga testen met het PSD van *Figuur 15*. Dit is expres een ingewikkeld PSD, om ingewikkelde problemen aan het licht te brengen.



Figuur 15 – Het PSD waarmee ik ga testen.

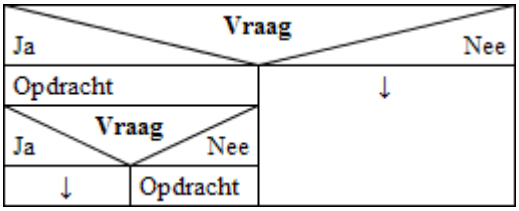
Tijdens het maken van dit PSD, ontdekte ik nog dat de pijlen in keuzeblokken niet goed gecentreerd worden. Dit is trouwens ook al te zien in *Figuur 14*, maar daar was het me nog niet opgevallen.
Mijn oude code voor het tekenen van de pijlen was (dit is voor een pijl in de ja-kolom):

```
    g.drawString("\u2193", ((xLinks+xRechts)/2-g.getFontMetrics().
                    stringWidth(blok.label)) / 2, hoogteOnderKeuzeblok+15);
```

Ik had deze code gekopieerd van het schrijven van het label van het blok. Daarbij had ik blijkbaar vergeten om de tweede keer `blok.label` te veranderen in `"\u2193"`, waardoor het uitlijnen verkeerd ging. De code moet dus zijn:

```
    g.drawString("\u2193", ((xLinks+xRechts)/2-g.getFontMetrics().
                      stringWidth("\u2193")) / 2, hoogteOnderKeuzeblok+15);
```

Hetzelfde geldt uiteraard ook voor de pijl in de nee-kolom. De pijlen worden nu wel goed uitgelijnd; zie *Figuur 16*.



Figuur 16 – De pijlen worden nu wel goed uitgelijnd

Blokken schuiven

Hierbij gaat het erom, dat het schuiven van grote, ingewikkelde blokken goed verloopt. Denk daarbij aan het verschuiven van een keuzeblok, waarin aan beide kanten weer veel geneste blokken staan.

- Als eerste heb ik het grote keuzeblok, dat als tweede blok in het PSD staat, verplaatst. Dat ging helemaal goed.
- Daarna heb ik het keuzeblok, dat aan de nee-kant van het grote keuzeblok staat, verplaatst. Dat ging helemaal goed.
- Ten slotte heb ik het herhalingsblok onderaan verplaatst. Ook dat ging helemaal goed.

Naar aanleiding van deze test heb ik nog wel twee verbeterpunten:

- Als je een blok "uit" het PSD schuift, dus als je het nog een keer omlaag schuift als het als onderste staat of het omhoog schuift als het als bovenste staat, dan wordt er een ArrayIndexOutOfBoundsException opgegooid:

```
Exception in thread "AWT-EventQueue-0"
java.lang.ArrayIndexOutOfBoundsException: Array index out of range:
35
        at java.util.Vector.get(Unknown Source)
        at PSD.blokkenInVectorOmwisselen(PSD.java:428)
        at PSD.verschuifBlokOmlaag(PSD.java:334)
        at ActieAfhandelaar.actionPerformed(ActieAfhandelaar.java:63)
              ...
```

  Dat is logisch, want er wordt dan geprobeerd om een index in de blokkenVector aan te spreken, die buiten het bereik van de Vector ligt. Overigens gaat de uitvoering van het programma trouwens goed verder, want de Exception zorgt er sowieso al voor dat de verplaatsing wordt afgebroken. Een simpele oplossing is dus om de fout af te vangen: in *verschuifBlokOmhoog*:

```
    try { blokkenInVectorOmwisselen(blokNummer, blokNummer-1); }
    catch (ArrayIndexOutOfBoundsException e) {
                  System.err.println("Omhoog verplaatsen mislukt."); }
```

  en op dezelfde manier in *verschuifBlokOmlaag*.
  Verder moeten *bepaalBlokLengte* en *verwijderBlok* op dezelfde manier worden aangepast, want ook deze kunnen een ArrayIndexOutOfBoundsException opgooien.
- Een blok dat wordt toegevoegd, zou eigenlijk beter kunnen worden toegevoegd *onder* de huidige selectie (momenteel wordt een nieuw blok toegevoegd aan het einde van het PSD). Dat heeft als voordeel dat je niet ieder blok een heel eind omhoog moet schuiven voordat het op de goede plaats staat.
  In de toevoeg-methoden vervang ik dus

```
blokkenVector.addElement(new Blok(soort blok, naam));
```

door

```
int toeTeVoegenPositie = (selectie == -1 ? blokkenVector.size() :
                                                    selectie + 1)
blokkenVector.add(toeTeVoegenPositie, new Blok(soort blok, naam));
```

Bovendien is het een goed idee als het nieuw toegevoegde blok direct wordt geselecteerd, zodat de gebruiker meteen daarna een blok daarachter kan zetten. De volgende regel wordt dus toegevoegd:

```
    selectie = toeTeVoegenPositie;
```

Blokken verwijderen

Ik heb een aantal blokken verwijderd; dit ging allemaal goed.

Blokken selecteren in ingewikkelde PSD's

Ik heb een aantal blokken geselecteerd; dit ging allemaal goed. Ook de herhalingsblokken zijn nu in orde (zie *Ingewikkelde PSD's testen*, pag. 30).

Stap 6: Documenteren

De code

De code is gedurende het programmeren al gedocumenteerd. Ook heb ik al Javadoc-annotaties geschreven. Javadoc heeft met behulp daarvan documentatie gegenereerd.
Verder heb ik de code in een jar-archief gezet. Hierdoor kan het programma onder Windows eenvoudig worden opgestart, door het jar-archief te openen. Het maken van een jar-archief gaat met het programmaatje *jar.exe* dat in de JDK zit.

Jar-archieven

Een jar-archief is een bestand met de extensie .jar. Het is eigenlijk gewoon een zip-archief waarin alle .class-bestanden staan en de bronnen die nodig zijn voor het laden van het programma (in het geval van de PSD Creator zijn dat bijvoorbeeld de iconen van de werkbalk).

Het gebruik van jar-archieven heeft twee grote voordelen: als eerste kun je geen bestanden die bij het programma horen, zoals de iconen, "kwijtraken", en als tweede kun je voor een jar-archief een *Main-Class*-attribuut instellen, dat aangeeft wat de hoofdklasse van het programma is. Als het jar-archief nu wordt geopend, dan wordt deze *Main-Class* aangeroepen, zodat het programma geopend wordt. Het voordeel daarvan is dat er door de gebruiker geen opdrachtprompt hoeft te worden geopend.

Gebruikershandleiding

De gebruikershandleiding is te vinden op de CD in de map *PSD Creator*.

Stap 2: Probleem analyseren (vervolg)

Algoritmen voor het PSD

De algoritmen voor het PSD (opslag / verplaatsen / selecteren / verwijderen van blokken, tekenen van het PSD) werden grotendeels besproken bij de PSD Creator.
Hieronder zal ik aangeven hoe ik de grote verscheidenheid aan blokken zal opslaan, want dat is een belangrijk verschil tussen de PSD Creator en de Applet Creator: de PSD Creator hoeft slechts 3 soorten blokken te onthouden (met verschillende vormen), terwijl de Applet Creator er 23 heeft.
In de tabs, aan de rechterkant van het scherm, worden blokken getekend met vooringestelde waarden voor

- de vorm van het blok
- het label op het blok
- de code die gegenereerd gaat worden als het blok wordt omgezet in Java-code
- bij GUI-blokken: de code die gegenereerd gaat worden *in de GUI-declaratie* als het blok wordt omgezet in Java-code
- bij GUI-blokken: de code die gegenereerd gaat worden *in de action-methode* als het blok wordt omgezet in Java-code
- de eigenschappen die er voor het blok kunnen worden ingesteld
- de waarden van die eigenschappen

Als de gebruiker nu op zo'n blok klikt, dan worden die waarden gekopieerd naar een instantie van de klasse Blok, die in de blokkenVector komt te staan. Een soort blok heeft dus niet een eigen klasse; alle blokken gebruiken gewoon de klasse Blok.
De klasse Blok heeft de volgende *public* variabelen:

- **blokVorm** komt overeen met *typeBlok* uit de PSD Creator; deze blijft gewoon Gewoon, Keuze, Herhaling, Eind Keuze, Herhaling of Ja-nee. Op die manier weet het tekenalgoritme welk soort blok er moet worden getekend.
- **label** blijft ook, maar met aanpassingen. Bij de PSD Creator werden er namelijk gewoon vaste Strings gegeven die op de blokken kwamen te staan. Bij de Applet Creator is er sprake van verschillende *soorten* blokken die allemaal een vastgestelde tekst op zich dragen, maar bepaalde details kunnen verschillen. Bijvoorbeeld, een herhalingsblok kan de ene keer *Herhaal 2 keer met teller teller* op zich hebben, en de andere keer *Herhaal 4 keer met teller tellerVariabele* of zelfs *Herhaal aantalElementen keer met teller* (daarin is *aantalElementen* een variabele).
  In **label** wordt bij dit herhalingsblok opgeslagen: *Herhaal {{{1}}} keer met teller {{{2}}}*. In *{{{1}}}* en *{{{2}}}* worden dan de eigenschappen van het blok ingevuld, zie **eigenschappenArray** hieronder.
- **javaCode**: de code die gegenereerd gaat worden als het blok wordt omgezet naar Java-code. Op dezelfde manier als **label**, bijvoorbeeld: *for ({{{2}}} = 0; {{{2}}} <= {{{1}}}; {{{2}}}++)*.
- **javaCodeGUI**: de code die in de GUI-declaratie gegenereerd gaat worden als het blok wordt omgezet naar Java-code. Voor een knop bijvoorbeeld: *Button {{{1}}};* (waarbij *{{{1}}}* de naam van de gemaakte knop is). Bij niet-GUI-blokken wordt deze variabele op null ingesteld.
- **javaCodeAction**: de code die in de action-methode gegenereerd gaat worden als het blok wordt omgezet naar Java-code. Voor een knop bijvoorbeeld: *if (e.target == {{{1}}}) { {{{2}}}(); return true; }* (waarbij *{{{2}}}* de naam van de uit te voeren methode is). Bij niet-GUI-blokken wordt deze variabele weer op null ingesteld.

- **eigenschappenArray**, dit is een array waarin alle eigenschappen van het blok worden opgeslagen. De eigenschappen van het blok zijn de teksten die omschrijven wat alle eigenschappen inhouden. Deze teksten worden bijvoorbeeld weergegeven in het blok-bewerken-venster. De array bevat Strings. Bij een herhalingsblok zou dit bijvoorbeeld zijn {“Hoeveel keer herhalen?”, “In welke variabele moet de teller worden bijgehouden?”}.
- **waardenVanEigenschappenArray**, dit is een array waarin alle waarden van de eigenschappen van het blok worden opgeslagen. De array bevat Strings, die de waarden geven. Bij het herhalingsblok hierboven zou dit bijvoorbeeld zijn {“2”, “teller”}, {“4”, “tellerVariabele”} of {“aantalElementen”, “teller”}.

Uitwerking van de benodigde blokken

Ik heb in Excel van ieder blok uitgewerkt welke waarden deze variabelen moeten hebben. Zie hiervoor *pad toevoegen*. Hieronder, in *Figuur 17*, staat ter referentie een afbeelding van dit werkblad.

Naam van blok	blokVorm	label	javaCode	javaCodeGUI	javaCodeAction	eigenschappenArray	waardenVanEigen-schappenArray
Herhaal een bepaald aantal keer	Herhaling	Herhaal {{{1}}} keer met teller {{{2}}}	`for ({{{2}}} = 0; {{{2}}} <= {{{1}}}; {{{2}}}++) {`	null	null	{"Hoeveel keer herhalen?", "In welke variabele moet de teller worden bijgehouden?"}	null
Herhaal zolang een voorwaarde waar is	Herhaling	Herhaal zolang {{{1}}}	`while ({{{1}}}) {`	null	null	{"Welke voorwaarde?"}	null
Keuze op basis van een vraag	Keuze	{{{1}}}?	`if ({{{1}}}) {`	null	null	{"Welke vraag?"}	null
Voer een methode uit	Gewoon	Voer methode {{{1}}} uit	`this.{{{1}}}({{{2}}});`	null	null	{"Welke methode?", "Welke parameters moeten er met deze methode worden meegegeven?"	null
Een waarde teruggeven	Gewoon	Geef terug: {{{1}}}	`return {{{1}}};`	null	null	{"Welke waarde moet worden teruggegeven? Let wel op: hij moet van het type zijn zoals beschreven in het methode-bewerken-venster."}	null
Stel een variabele in	Gewoon	Stel variabele {{{1}}} in op {{{2}}}	`{{{1}}} = {{{2}}};`	null	null	{"Welke variabele?", "Welke waarde?"}	null
Verhoog een variabele met 1	Gewoon	Verhoog variabele {{{1}}}	`{{{1}}}++;`	null	null	{"Welke variabele?"}	null
Verlaag een variabele met 1	Gewoon	Verlaag variabele {{{2}}}	`{{{2}}}--;`	null	null	{"Welke variabele?"}	null
Schrijf tekst op het scherm	Gewoon	Schrijf {{{1}}} op ({{{2}}}, {{{3}}})	`pen.drawString({{{1}}}, {{{2}}}, {{{3}}});`	null	null	{"Welke tekst? Let op: als je letterlijke tekst wilt schrijven, zet er dan dubbele aanhalingstekens omheen. Wil je een variabele schrijven, zet er dan geen aanhalingstekens omheen. Stukken tekst kun je aan elkaar “lijmen” met een plusteken.", "Wat is de x-coördinaat?", "Wat is de y-coördinaat?"}	null
Teken een lijn	Gewoon	Teken lijn van ({{{1}}}, {{{2}}}) tot ({{{3}}}, {{{4}}})	`pen.drawLine({{{1}}}, {{{2}}}, {{{3}}}, {{{4}}});`	null	null	{"x-coördinaat beginpunt?", "y-coördinaat beginpunt", "x-coördinaat eindpunt", "y-coördinaat eindpunt"}	null
Teken een rechthoek	Gewoon	Teken rechthoek van ({{{1}}}, {{{2}}}) tot ({{{3}}}, {{{4}}})	`pen.drawRect({{{1}}}, {{{2}}}, {{{3}}}, {{{4}}});`	null	null	{"x-coördinaat linksbovenhoek?", "y-coördinaat linksbovenhoek?", "Breedte?", "Hoogte?"}	null
Teken een gevulde rechthoek	Gewoon	Teken gevulde rechthoek van ({{{1}}}, {{{2}}}) tot ({{{3}}}, {{{4}}})	`pen.fillRect({{{1}}}, {{{2}}}, {{{3}}}, {{{4}}});`	null	null	{"x-coördinaat linksbovenhoek?", "y-coördinaat linksbovenhoek?", "Breedte?", "Hoogte?"}	null
Teken een ovaal	Gewoon	Teken ovaal van ({{{1}}}, {{{2}}}) tot ({{{3}}}, {{{4}}})	`pen.drawOval({{{1}}}, {{{2}}}, {{{3}}}, {{{4}}});`	null	null	{"x-coördinaat linksbovenhoek?", "y-coördinaat linksbovenhoek?", "Breedte?", "Hoogte?"}	null
Teken een gevulde ovaal	Gewoon	Teken gevulde ovaal van ({{{1}}}, {{{2}}}) tot ({{{3}}}, {{{4}}})	`pen.fillOval({{{1}}}, {{{2}}}, {{{3}}}, {{{4}}});`	null	null	{"x-coördinaat linksbovenhoek?", "y-coördinaat linksbovenhoek?", "Breedte?", "Hoogte?"}	null
Verander de tekenkleur	Gewoon	Verander de tekenkleur in {{{1}}}	`pen.setColor({{{1}}});`	null	null	{"Geef de nieuwe kleur. Kies uit: black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, yellow."}	null
Maak een knop (Button)	Gewoon	Maak de knop {{{1}}}	`{{{1}}} = new Button({{{2}}}); add({{{1}}});`	`Button {{{1}}};`	`if (e.target == {{{1}}}) { this.{{{2}}}(); }`	{"Wat is de naam van de knop?", "Wat is het opschrift van de knop?", "Welke methode moet worden uitgevoerd, als er op de knop wordt geklikt?"	null
Maak een invoervak (TextField)	Gewoon	Maak het invoervak {{{1}}}	`{{{1}}} = new TextField({{{2}}}); add({{{1}}});`	`TextField {{{1}}};`	null	{"Wat is de naam van het vakje?", "Wat is de breedte van het vakje (uitgedrukt in het aantal letters, bijv. 20)?"}	null
Maak een vast tekstvak (Label)	Gewoon	Maak het vaste tekstvak {{{1}}}	`{{{1}}} = new Label({{{2}}}); add({{{1}}});`	`Label {{{1}}};`	null	{"Wat is de naam van het tekstvak?", "Welke tekst moet erin?"}	null
Stel de positie van een element in	Gewoon	Zet {{{1}}} op ({{{2}}}, {{{3}}}) met formaat {{{4}}}x{{{5}}}	`{{{1}}}.setBounds({{{2}}}, {{{3}}}, {{{4}}}, {{{5}}});`	null	null	{"Van welk element?", "x-coördinaat linksbovenhoek?", "y-coördinaat linksbovenhoek?", "Breedte?", "Hoogte?"}	null
De tekst op een knop aanpassen	Gewoon	Tekst op {{{1}}} veranderen in {{{2}}}	`{{{1}}}.setLabel({{{2}}});`	null	null	{"Van welke knop?", "Wat is de nieuwe tekst?"}	null
De tekst in een invoervak aanpassen	Gewoon	Tekst in {{{1}}} veranderen in {{{2}}}	`{{{1}}}.setText({{{2}}});`	null	null	{"Van welk invoervak?", "Wat is de nieuwe tekst?"}	null
De tekst in een vast tekstvak aanpassen	Gewoon	Tekst van {{{1}}} veranderen in {{{2}}}	`{{{1}}}.setText({{{2}}});`	null	null	{"Van welk tekstvak?", "Wat is de nieuwe tekst?"}	null

Figuur 17 – Excel-werkblad met de blokken erin

Het uitvoeralgoritme

Het uitvoeralgoritme is heel ingewikkeld, omdat er meerdere methoden zijn die allemaal meerdere blokken bevatten. Bovendien zijn er nog de standaardmethoden, die het allemaal nog ingewikkelder maken.
Om het uitvoeralgoritme te structureren, maak ik gebruik van de analyse van de standaardcode (zie pag. 13).
Het hoofdalgoritme is:

genereerCode()

- *inspringing* = 0
- Schrijf: “import java.awt.*;”
- Schrijf: “import java.applet.*;”
- Schrijf: “” *(lege regel)*
- Schrijf: “public class ” + de ingegeven bestandsnaam + “ extends Applet”
- Schrijf: “{”
- Vergroot *inspringing* met 4
- **genereerCodeInitialisatie()** ← geeft codeVanInit, GUIVariabeleDeclaratie en codeVanAction terug
- **vectorSchrijven(**GUIVariabeleDeclaratie**)**
- Schrijf: “public void init()”
- Schrijf: “{”
- Vergroot *inspringing* met 4
- **vectorSchrijven(**codeVanInit**)**
- Verklein *inspringing* met 4
- Schrijf: “}”
- Schrijf: “” *(lege regel)*
- **genereerCodeTekenmethode()**← geeft codeVanPaint terug
- Schrijf: “public void paint(Graphics pen)”
- Schrijf: “{”
- Vergroot *inspringing* met 4
- **vectorSchrijven(**codeVanPaint**)**
- Verklein *inspringing* met 4
- Schrijf: “}”
- Schrijf: “” *(lege regel)*
- Schrijf: “public boolean action(Event e, Object o)”
- Schrijf: “{”
- Vergroot *inspringing* met 4
- **vectorSchrijven(**codeVanAction**)**
- Verklein *inspringing* met 4
- Schrijf: “}”
- Schrijf: “” *(lege regel)*
- **genereerCodeAndereMethoden()**← geeft codeVanAndereMethoden terug
- **vectorSchrijven(**codeVanAndereMethoden**)**
- Verklein *inspringing* met 4
- Schrijf: “}”

Deze methode roept weer andere methoden aan, namelijk:

- **genereerCodeInitialisatie()**: deze methode genereert de code voor de init-methode. Deze code wordt weggeschreven naar een Vector: één String in de Vector per regel.

Verder worden er Vectoren gemaakt met de variabelen die door de GUI worden aangemaakt, bijvoorbeeld:

```
Button mijnKnopje;
```

en er wordt een Vector aangemaakt met de action-methode van de applet erin, bijvoorbeeld:

```
if (e.target == mijnKnopje)
{
    // doe iets
    return true;
}
return false;
```

- **genereerCodeTekenmethode()**: deze methode genereert de code voor de paint-methode. Deze code wordt weer weggeschreven naar een Vector.
- **genereerCodeAndereMethoden()**: deze methode genereert de code voor alle andere methoden, inclusief methode-headers. Een voorbeeld is:

```
public void doeIets()
{
    // code
}

public int berekenKwadraat(int invoer)
{
    // code
}
```

- **vectorSchrijven(Vector)**: schrijft alle Strings in de Vector naar het uitvoerbestand.

Hieronder worden de algoritmen voor deze methoden uitgewerkt.

genereerCodeInitialisatie()

- Voor iedere variabele die in het methodevenster van Initialiseren gegeven is:
  - Haal naam en soort van de variabele op
  - Schrijf *(naar Vector codeVanInit)*: soort + " " + naam + ";"
- Voor ieder blok in de blokkenVector van Initialiseren:
  - Haal *javaCode* en *waardenVanEigenschappenArray* van het blok op
  - Vervang alle {{{1}}} in *javaCode* door *waardenVanEigenschappenArray[1]*
  - Doe hetzelfde voor {{{2}}}, enzovoorts
  - Schrijf *(naar Vector codeVanInit)*: de verkregen code
  - Als *javaCodeGUI* van het blok niet *null* is, dan
    - Vervang alle {{{1}}} in *javaCodeGUI* door *waardenVanEigenschappenArray[1]*
    - Doe hetzelfde voor {{{2}}}, enzovoorts
    - Schrijf *(naar Vector GUIVariabeleDeclaratie)*: de verkregen code
  - Als *javaCodeAction* van het blok niet *null* is, dan
    - Vervang alle {{{1}}} in *javaCodeAction* door *waardenVanEigenschappenArray[1]*
    - Doe hetzelfde voor {{{2}}}, enzovoorts
    - Schrijf *(naar Vector codeVanAction)*: de verkregen code

genereerCodeTekenmethode()

- Voor iedere variabele die in het methodevenster van Tekenmethode gegeven is:
  - Haal naam en soort van de variabele op
  - Schrijf *(naar Vector codeVanPaint)*: soort + " " + naam + ";"
- Voor ieder blok in de blokkenVector van Tekenmethode:
  - Haal *javaCode* en *waardenVanEigenschappenArray* van het blok op
  - Vervang alle {{{1}}} in *javaCode* door *waardenVanEigenschappenArray[1]*

    - Doe hetzelfde voor {{{2}}}, enzovoorts
    - Schrijf *(naar Vector codeVanPaint)*: de verkregen code

genereerCodeAndereMethoden()

- Voor iedere methode, behalve voor de standaardmethoden:
    - Schrijf *(naar Vector codeVanAndereMethoden)*: publicOfPrivate + returnType + methodeNaam + "(" + methodeParameters + ")"
    - Schrijf "{"
    - Vergroot *inspringing* met 4
    - Voor iedere variabele die in het methodevenster van die methode gegeven is:
        - Haal naam en soort van de variabele op
        - Schrijf *(naar Vector codeVanAndereMethoden)*: soort + " " + naam + ";"
    - Voor ieder blok in de blokkenVector van die methode:
        - Haal *javaCode* en *waardenVanEigenschappenArray* van het blok op
        - Vervang alle {{{1}}} in *javaCode* door *waardenVanEigenschappenArray[1]*
        - Doe hetzelfde voor {{{2}}}, enzovoorts
        - Schrijf *(naar Vector codeVanAndereMethoden)*: de verkregen code
    - Verklein *inspringing* met 4
    - Schrijf "}"

vectorSchrijven(Vector)

- Voor ieder element in de Vector
    - Herhaal *inspringing* keer
        - Schrijf: een spatie
    - Schrijf: het element in de Vector
    - Schrijf: een regeleinde

Stap 3: Oplossing schematiseren

In deze stap ga ik PSD's maken om de algoritmen weer te geven. Uiteraard maak ik de PSD's met de PSD Creator. Eigenlijk is het uitvoeralgoritme het enige PSD dat ik ga maken, want alle andere PSD's zijn al gemaakt bij de PSD Creator.

Het uitvoeralgoritme

genereerCode()

Dit is een vrij saai PSD; het wordt hier gegeven voor de volledigheid:

```
inspringing = 0
schrijf("import java.awt.*;")
schrijf("import java.applet.*;")
schrijf("")
schrijf("public class " + bestandsnaam + " extends Applet")
schrijf("{")
inspringing += 4
genereerCodeInitialisatie()
vectorSchrijven(GUIVariabeleDeclaratie)
schrijf("public void init()")
schrijf("{")
inspringing += 4
vectorSchrijven(codeVanInit)
inspringing -= 4
schrijf("}")
schrijf("")
genereerCodeTekenmethode()
schrijf("public void paint(Graphics pen)")
schrijf("{")
inspringing += 4
vectorSchrijven(codeVanPaint)
inspringing -= 4
schrijf("}")
schrijf("")
schrijf("public boolean action(Event e, Object o)")
schrijf("{")
inspringing += 4
vectorSchrijven(codeVanAction)
inspringing -= 4
schrijf("}")
schrijf("")
genereerCodeAndereMethoden()
vectorSchrijven(codeVanAndereMethoden)
inspringing -= 4
schrijf("}")
```

genereerCodeInitialisatie()

Maak de Vectoren codeVanInit, GUIVariabeleDeclaratie en codeVanAction leeg

Herhaal voor iedere variabele die in het methodevenster van Initialiseren gegeven is

- Haal naam en soort van de variabele op
- schrijfNaarVector(codeVanInit, soort + " " + naam + ";")

Herhaal voor ieder blok in de blokkenVector van Initialiseren

- Haal van het blok op: javaCode, waardenVanEigenschappenArray, javaCodeGUI, javaCodeAction
- **Herhaal voor teller van 1 t/m 5**
  - Vervang alle {{{teller}}} in javaCode door waardenVanEigenschappenArray[teller]
- schrijfNaarVector(codeVanInit, [de verkregen code])

javaCodeGUI != null?

Ja:

- **Herhaal voor teller van 1 t/m 5**
  - Vervang alle {{{teller}}} in javaCodeGUI
- schrijfNaarVector(GUIVariabeleDeclaratie, [code])

Nee: ↓

javaCodeAction != null?

Ja:

- **Herhaal voor teller van 1 t/m 5**
  - Vervang alle {{{teller}}} in javaCodeAction
- schrijfNaarVector(codeVanAction, [code])

Nee: ↓

genereerCodeTekenmethode()

Maak de Vector codeVanPaint leeg

Herhaal voor iedere variabele die in het methodevenster van Tekenmethode gegeven is

- Haal naam en soort van de variabele op
- schrijfNaarVector(codeVanInit, soort + " " + naam + ";")

Herhaal voor ieder blok in de blokkenVector van Tekenmethode

- Haal van het blok op: javaCode, waardenVanEigenschappenArray
- **Herhaal voor teller van 1 t/m 5**
  - Vervang alle {{{teller}}} in javaCode door waardenVanEigenschappenArray[teller]
- schrijfNaarVector(codeVanPaint, [de verkregen code])

genereerCodeAndereMethoden()

Herhaal voor iedere methode, behalve de standaardmethoden

- schrijfNaarVector(codeVanAndereMethoden, publicOfPrivate + " " + returnType + " " + methodeNaam + "(" + methodeParameters + ")")
- schrijfNaarVector(codeVanAndereMethoden, "{")
- inspringing += 4
- **Herhaal voor iedere variabele die in het methodevenster van die methode gegeven is**
  - Haal naam en soort van de variabele op
  - schrijfNaarVector(codeVanAndereMethoden, soort + " " + naam + ";")
- **Herhaal voor ieder blok in de blokkenVector van die methode**
  - Haal van het blok op: javaCode, waardenVanEigenschappenArray
  - **Herhal voor teller van 1 t/m 5**
    - Vervang alle {{{teller}}} in javaCode door waardenVanEigenschappenArray[teller]
  - schrijfNaarVector(codeVanAndereMethoden, [de verkregen code])
- inspringing -= 4
- schrijfNaarVector(codeVanAndereMethoden, "}")
- schrijfNaarVector(codeVanAndereMethoden, "")

schrijfNaarVector(Vector, String)

Maak een nieuwe, lege String teSchrijvenString
Herhaal inspringing keer
teSchrijvenString = teSchrijvenString + " "
teSchrijvenString = teSchrijvenString + String [die is meegegeven met de methode]
teSchrijvenString = teSchrijvenString + [regeleinde]
Schrijf teSchrijvenString als laatste element in de Vector

vectorSchrijven(Vector)

Herhaal voor ieder element in de Vector
schrijf([Dat element])

schrijf(String)

Maak een nieuwe, lege String teSchrijvenString
Herhaal inspringing keer
teSchrijvenString = teSchrijvenString + " "
teSchrijvenString = teSchrijvenString + String [die is meegegeven met de methode]
teSchrijvenString = teSchrijvenString + [regeleinde]
Schrijf teSchrijvenString naar de uitvoer

Stap 4: Programma-instructies coderen

In deze stap wordt de Applet Creator echt gemaakt. Als eerste ga ik de indeling in klassen bedenken voor het programma. Daarna ga ik de code schrijven: ik begin bij de GUI (Graphical User Interface); daarna ga ik verder met de werking van het programma.

De klassenindeling

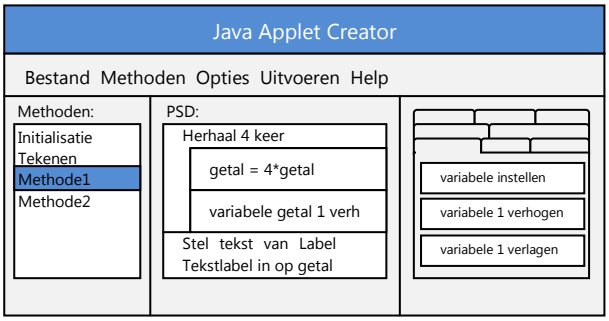
Als eerste zal ik aangeven welke klassen er nodig zijn voor de GUI, daarna volgen de klassen die nodig zijn voor de opslag van PSD en methoden.

Klassen voor de GUI (Graphical User Interface)

De hoofdklasse is **AppletCreator**. Deze klasse maakt het hoofdvenster aan. AppletCreator maakt instanties aan van HoofdMenubalk en HoofdContentPane, om respectievelijk een menubalk en een contentpane (alles in een venster, behalve de menubalk) te krijgen.

De klasse voor de menubalk is, zoals gezegd, **HoofdMenubalk**. Deze klasse *extends* de klasse JMenuBar van Java Swing, om op een vrij eenvoudige manier een menubalk in elkaar te zetten. Deze klasse maakt een instantie van ActieAfhandelaar (zie beneden).

De klasse voor de contentpane is **HoofdContentPane**. Deze klasse zorgt voor de indeling van het scherm. HoofdContentPane roept erg veel klassen aan (zie *Figuur 19* voor een grafische weergave), namelijk:



Figuur 18 – Nogmaals het basisontwerp voor de GUI (uit stap 1)

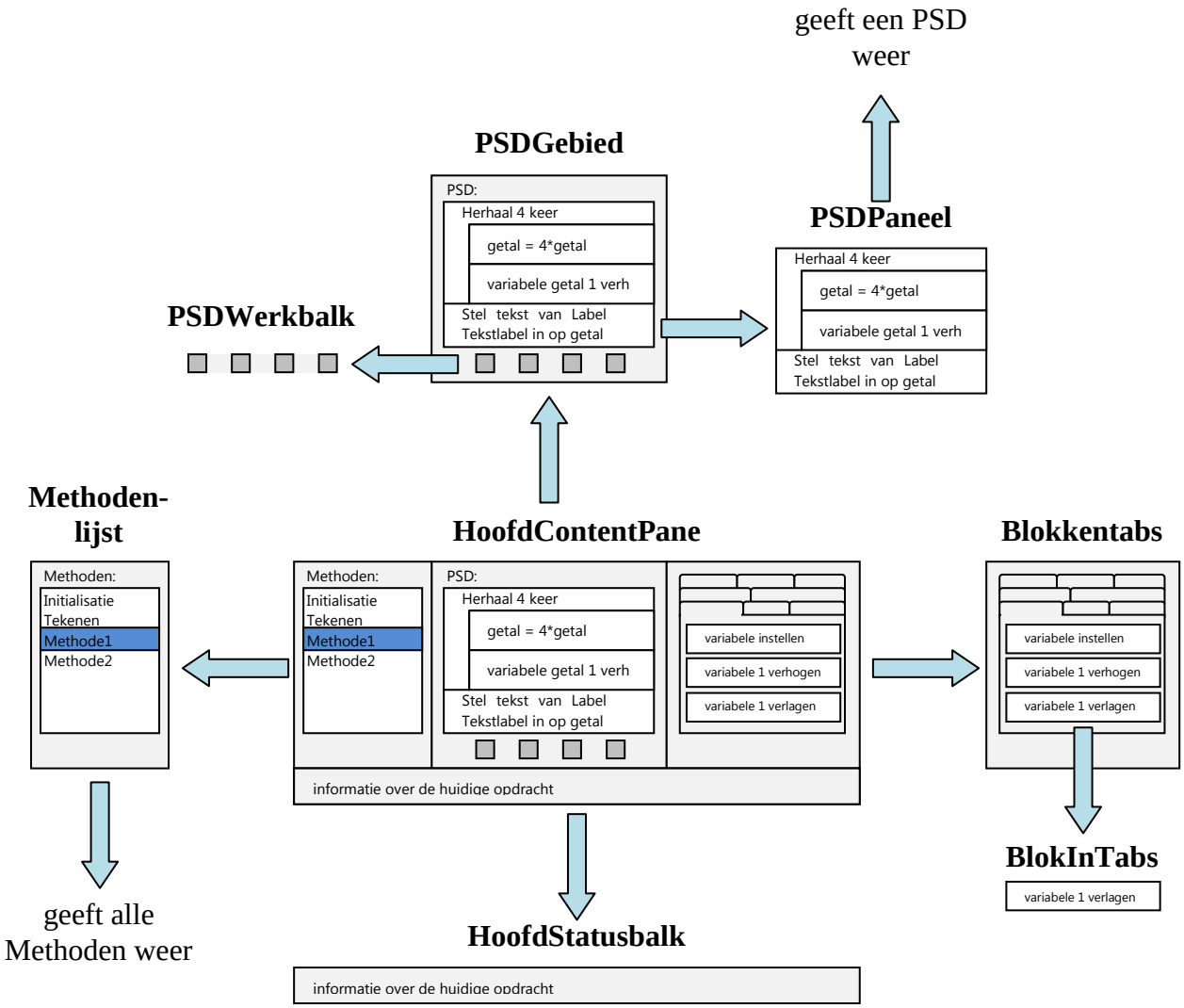
- **HoofdStatusbalk**: een simpel JLabel met methoden om de tekst in de statusbalk aan te passen.
- **Methodenlijst**: dit is een JPanel met daarin een JList die alle methoden weergeeft. Boven de JList staat een JLabel.
- **PSDGebied**: dit is een JPanel. Aan de bovenkant van dit JPanel staat een JLabel, in het midden staat een PSDPaneel (zie beneden) en onderaan staat een kleine werkbalk om blokken te verplaatsen en te verwijderen: een PSDWerkbalk (zie beneden).
- **PSDPaneel**: dit is het JPanel waarop het PSD getekend wordt. In de PSD Creator komt dit overeen met PSD. Een groot verschil tussen de PSD Creator en de Applet Creator is dat in de PSD Creator PSD ook zelf de blokkenVector bevat, en in de Applet Creator staat deze in een Methode.
- **PSDWerkbalk**: een kleine werkbalk die in PSDGebied staat en die de gebruiker in staat stelt om blokken te verplaatsen en te verwijderen. Dit is een JToolBar. PSDWerkbalk maakt een ActieAfhandelaar aan.
- **Blokkentabs**: de tabbladen met daarin de blokken die in het PSD kunnen worden geplaatst. Blokkentabs maakt **BlokInTabs**-instanties aan, die de blokken in de tabs voorstellen. BlokInTabs is een simpel JPanel.

De klasse **ActieAfhandelaar** zorgt voor de afhandeling van events die optreden vanuit de HoofdMenubalk en de PSDWerkbalk. ActieAfhandelaar kijkt welke actie er heeft plaatsgevonden en beslist met een serie if-constructies wat er moet worden gedaan.

Er zijn ook nog klassen voor aparte vensters.

- De klasse **BlokBewerkenVenster** geeft alle eigenschappen van het blok weer en laat de gebruiker die aanpassen.

- De klasse **MethodeBewerkenVenster**, dit is de klasse die het venster weergeeft waarmee een methode wordt bewerkt. Aangezien van Initialisatie en Tekenmethode alleen de variabelen kunnen worden bewerkt, is **StandaardMethodeBewerkenVenster** een speciaal venster voor het bewerken van die methoden.
- De klasse **OptiesVenster** zorgt ervoor dat de gebruiker de locatie van *javac.exe* en *appletviewer.exe* kan aanpassen.
- De klasse **Helpvenster** geeft de helpinformatie weer over het programma.
- De klasse **Aboutbox** geeft de versie van het programma weer.



Figuur 19 – Grafische weergave van de klassen van de GUI die HoofdContentPane aanroept. Een pijl betekent: deze klasse maakt een instantie van de klasse waar de pijl naartoe wijst. HoofdContentPane maakt bijvoorbeeld een instantie van HoofdStatusbalk aan om de statusbalk te tekenen.

Klassen voor de opslag van het PSD

AppletCreator maakt als eerste een instantie van de klasse **MethodeVerzameling** aan. Deze klasse bevat een Vector met daarin de methoden.

Al deze methoden bevatten een blokkenVector met daarin de blokken. Als er nu geklikt wordt op een andere methode in Methodenlijst, dan wordt de juiste methode ingeladen in het PSDGebied.
De blokkenVector bevat, net zoals bij de PSD Creator, een blokkenVector die de **Blok**ken opslaat. Een Blok is een simpele klasse met de variabelen *blokVorm*, *label*, *javaCode*, *javaCodeGUI*, *javaCodeAction*, *eigenschappenArray* en *waardenVanEigenschappenArray*, zoals beschreven in Algoritmen voor het PSD (pag. 35).

Klassen voor de uitvoer

Voor de uitvoer weet ik nog niet hoe ik daar de klassen ga indelen. Ik zal daar later (op pagina 60) op terugkomen.

De GUI maken

Nu ga ik de klassen schrijven. Bij iedere klasse staat een beknopte uitleg over de klasse en een overzicht van de leden (variabelen, constructors en methoden) van de klasse. Bij de variabelen worden meestal alleen de andere klassen genoemd die geïnstantieerd worden.

Klasse AppletCreator

Dit is een standaardklasse om een programma te laden. Hij komt voor een groot deel overeen met de klasse PSDCreator van de PSD Creator. Als eerste wordt er overgestapt op de Event Dispatching Thread. In deze thread wordt de GUI opgebouwd.

De Event Dispatching Thread
De Event Dispatching Thread (EDT) is de thread die door Swing wordt gebruikt om de GUI op te bouwen. Als er een andere thread gebruikt wordt, dan kan dat concurrency-problemen opleveren, zoals deadlock.

Het opbouwen van de GUI bestaat uit het instellen van de content-pane en de menubalk.
Verder maakt AppletCreator een MethodeVerzameling aan.

Ledenoverzicht **AppletCreator**	
Variabelen / Velden	
public static MethodeVerzameling **hoofdMethodeVerzameling**	De verzameling van methoden van het hele programma.
Methoden	
public static void **main**(String[])	Methode waarmee het programma gestart wordt.
private static void **GUIMaken**()	Maakt de GUI van het programma. Mag alleen vanaf de EDT worden aangeroepen.

Klasse HoofdMenubalk

Dit is een JMenuBar. De opbouw is als volgt:

- Bestand
  - Nieuw
  - Openen…
  - Opslaan…
  - Afsluiten
- Methoden
  - Methode toevoegen…
  - Methode bewerken…
  - Methode verwijderen

- Opties
  - Opties…
- Uitvoeren
  - Java-code genereren…
  - Java-code genereren en uitvoeren…
- Help
  - Help
  - Info

Er wordt ook een ActieAfhandelaar aangemaakt om de knopklikken te registreren en af te handelen.

Ledenoverzicht **HoofdMenubalk**	
Variabelen / Velden	
ActieAfhandelaar **actie**	De ActieAfhandelaar die de events van deze menubalk afhandelt.
Constructoren	
public **HoofdMenubalk**()	Deze constructor laadt alle elementen in de menubalk.

Klasse HoofdContentPane

Dit is een JPanel. Hij gebruikt een BorderLayout als layout-manager. Dat betekent dat Methodenlijst, Blokkentabs en HoofdStatusbalk de benodigde, vastgestelde ruimte krijgen en het PSDGebied alle overige beschikbare ruimte krijgt. Als het venster dus groter wordt gemaakt, dan wordt PSDGebied groter.

Ledenoverzicht **HoofdContentPane**	
Variabelen / Velden	
public static Methodenlijst **methodenlijst**	De Methodenlijst die links in de contentpane komt te staan.
public static PSDGebied **psdGebied**	Het PSDGebied dat in het midden in de contentpane komt te staan.
public static Blokkentabs **blokkentabs**	De Blokkentabs die rechts in de contentpane komen te staan.
public static HoofdStatusbalk **statusbalk**	De statusbalk die onderin de contentpane komt te staan.
Constructoren	
public **HoofdContentPane**()	Deze constructor laadt de elementen in de content-pane volgens de BorderLayout.

Klasse HoofdStatusbalk

De statusbalk is bedoeld om korte informatie aan de gebruiker te geven. Deze klasse heeft twee methoden om de tekst daarop aan te passen. Eigenlijk zijn die overbodig (je kunt ook *getText* en *setText* gebruiken, omdat HoofdStatusbalk eigenlijk gewoon een JLabel is), maar misschien wil ik later dat er anders met de tekst omgegaan wordt en dan hoef ik niet overal de referenties aan te passen. Er is wel een probleem met de statusbalk: hij werkt niet goed tijdens het uitvoeren van een methode. Pas aan het einde van de methode wordt de tekst bijgewerkt.

Ledenoverzicht **HoofdStatusbalk**	
Constructoren	
public **HoofdStatusbalk**()	Maakt de statusbalk en stelt de grootte in.
Methoden	
public void **veranderStatusTekst**(String)	Verandert de statustekst in de statusbalk.
public String **geefStatusTekst**()	Geeft de statustekst die momenteel in de statusbalk staat.

Klasse Methodenlijst

Methodenlijst is een JPanel. Dit JPanel bevat in een BorderLayout een JLabel en een JScrollPane met daarin de JList waarin de methodennamen staan. De JList gebruikt een DefaultListModel om de namen te bewaren. Het voordeel daarvan is dat de JList automatisch aangepast wordt, als het DefaultListModel wordt gewijzigd.
Methodenlijst heeft enkele methoden om Strings toe te voegen, te bewerken en te verwijderen; deze kunnen dan bijvoorbeeld worden aangeroepen door de ActieAfhandelaar als er een nieuwe methode wordt toegevoegd.

Ledenoverzicht **Methodenlijst**	
Variabelen / Velden	
DefaultListModel **methodenNamen**	Een ListModel dat de naam van de methode bewaart. Op die manier kan de JList eenvoudig worden bijgewerkt.
Constructoren	
public **Methodenlijst**()	Laadt de elementen in en zet in *methodenNamen* alvast Initialiseren en Tekenmethode.
Methoden	
public void **voegNieuwElementToe**(String)	Voegt een nieuwe methodenaam toe aan de onderkant van de JList.
public void **bewerkElement**(int, String)	Vervangt de huidige String die op de opgegeven index staat door het nieuwe exemplaar dat meegegeven wordt.
public void **verwijderElement**(int)	Verwijdert de methodenaam op de meegegeven index uit de JList.
public int **geefSelectie**()	Geeft de geselecteerde index uit de lijst.

Klasse PSDGebied

Ook PSDGebied is een JPanel met BorderLayout. Bovenaan staat weer een JLabel. In het midden staat een instantie van PSDPaneel. Deze heeft een constructor die een PSD vraagt dat getekend moet worden. De constructor wordt als volgt aangeroepen:

```
Methode teBewerkenMethode =
                AppletCreator.hoofdMethodeVerzameling.geefMethode(0);
psdPaneel = new PSDPaneel(teBewerkenMethode);
add(new JScrollPane(psdPaneel), BorderLayout.CENTER);
```

Wat gebeurt hier nu? Als eerste wordt de Initialiseren-methode opgehaald uit de hoofdMethodeVerzameling. Daarna wordt de constructor aangeroepen met deze methode als argument. Ten slotte wordt aan het PSDGebied een scrollpane toegevoegd, met daarin het gemaakte PSDPaneel.

Ledenoverzicht **PSDGebied**	
Variabelen / Velden	
public PSDPaneel **psdPaneel**	Het PSDPaneel dat weergegeven wordt in dit PSDGebied.
Constructoren	
public **PSDGebied**()	Laadt de elementen in in de BorderLayout. In het bijzonder wordt het PSDPaneel ingeladen; erin staat dan het PSD van Initialiseren.

Klasse PSDPaneel

Dit is een ingewikkelde klasse van maar liefst 32 kB aan code! Het is een JPanel die het PSD tekent. De klasse is vrijwel rechtstreeks overgenomen van de PSD Creator-klasse PSD. Zie

hiervoor pagina 26 en verder. Het grote verschil is dat PSD een blokkenVector tekende (en bewerkte) die binnen de klasse lag opgeslagen; PSDPaneel tekent een blokkenVector van een bepaalde methode die wordt meegegeven in de constructor. Voor de rest is de klasse hetzelfde als PSD, met de volgende veranderingen:

- de nieuwe methode veranderTeTekenenPSD(Methode nieuweMethode), die een nieuw PSD inleest en tekent
- sommige variabelenamen zijn anders
- de toevoeg-methoden zijn veranderd in één toevoegmethode: voegBlokToe(Blok blok) waarbij de aanroeper een kant-en-klaar Blok moet opgeven
- de nieuwe methode veranderSelectie(int nieuweSelectie)

PSDPaneel werkt met het feit dat er, als er een constructor wordt aangeroepen, alleen een referentie naar het object wordt opgeslagen. Daarom kan PSDPaneel de Vector, die in de Methode zit die wordt meegegeven in de constructor, gewoon in teTekenenBlokkenVector zetten. Hier zit dan dus dezelfde referentie in als de Vector uit Methode. PSDPaneel kan dan dus door de gehele klasse heen teTekenenBlokkenVector aanroepen, terwijl eigenlijk de Vector in Methode wordt aangepast. Dat is precies de bedoeling, want teTekenenBlokkenVector kan dan makkelijk worden aangepast.

Geeft Java alleen referenties mee?

Wat hierboven staat, namelijk dat PSDPaneel de referentie naar de juiste Vector in teTekenenBlokkenVector kan plaatsen en dan de goede Vector bewerkt, wist ik niet zeker. Daarom heb ik dat getest. Zie de map *referentieTest* in de hoofdmap van Applet Creator. Voer het commando **java HoofdKlasse** uit om de test te openen. Er gebeurt het volgende:

1. HoofdKlasse maakt een nieuwe EersteKlasse aan.
2. EersteKlasse plaatst een element in een nieuw aangemaakte Vector en laat de Vector zien op de opdrachtprompt.
3. EersteKlasse maakt een nieuwe TweedeKlasse aan.
4. EersteKlasse roept een methode aan in TweedeKlasse en geeft de Vector als parameter mee. De methode in TweedeKlasse zet in de meegegeven Vector nog een element.
5. EersteKlasse laat de Vector weer zien op de opdrachtprompt.

Het is te zien dat de tweede keer dat EersteKlasse de Vector laat zien, er inderdaad het element bijstaat dat er door TweedeKlasse bij is gezet. Java geeft dus de referentie naar het object mee als parameter in een methode. Als dat niet zo zou zijn geweest, dus als Java het volledige object zou hebben doorgegeven, dan zou EersteKlasse tweemaal dezelfde Vector weergeven, want de aanroepende klasse zou dan nog steeds het oude object hebben.

Ledenoverzicht **PSDPaneel**	
Variabelen / Velden	
int **hoogte**	Hoogte die gebruikt wordt bij het tekenen en zoeken naar de muisklik.
int **selectie**	Het momenteel geselecteerde blok. -1 betekent: niets geselecteerd.
Vector **teTekenenBlokkenVector**	De blokkenvector die op dit moment getekend en bewerkt wordt.
Constructoren	
public **PSDPaneel**(Methode)	Maakt een nieuw PSDPaneel. Uit de opgegeven Methode wordt de blokkenVector gehaald en deze wordt dan gezet op *teTekenenBlokkenVector*.

Methoden	
public void **paintComponent**(Graphics)	Wordt automatisch aangeroepen als PSDPaneel opnieuw moet worden getekend.
int **tekenPSD**(int, int, int, Graphics)	Wordt aangeroepen door *paintComponent* en tekent het PSD in het PSDPaneel.
public void **voegToeBlok**(Blok)	Voegt het meegegeven blok toe op de juiste plaats in de *blokkenVector*. Als het blok een herhalingsblok is, dan wordt er automatisch een Eind Herhaling achter gezet. Als het blok een keuzeblok is, dan wordt er automatisch een Ja-nee en een Eind Keuze achter gezet.
public void **verschuifBlokOmlaag**(int)	Verschuift het blok op de meegegeven index in de *teTekenenBlokkenVector* één stap omlaag.
public void **verschuifBlokOmhoog**(int)	Verschuift het blok op de meegegeven index in de *teTekenenBlokkenVector* één stap omhoog.
int **bepaalBlokLengte**(int)	Bepaalt de lengte van het blok op de meegegeven index in de *teTekenenBlokkenVector*.
void **blokkenInVectorOmwisselen**(int, int)	Wisselt de twee opgegeven blokken om in de *teTekenenBlokkenVector*.
public int **zoekGekliktBlok**(int, int, int, int, int)	Zoekt op welk blok geklikt is en stelt daar de selectie op in.
public void **verwijderBlok**(int)	Verwijdert het blok op de meegegeven index in de *teTekenenBlokkenVector*. Subblokken worden ook verwijderd.
public void **mousePressed**(MouseEvent) *(en de drie volgende)*	Hoort bij MouseListener, maar wordt niet gebruikt.
public void **mouseClicked**(MouseEvent)	Hoort bij MouseListener; wordt automatisch aangeroepen als er op een blok geklikt wordt. Deze methode roept *zoekGekliktBlok* aan.
public void **veranderTeTekenenPSD**(Methode)	Laadt de blokkenVector van de meegegeven methode in in de *teTekenenBlokkenVector*, zodat het PSD van die andere methode getekend en bewerkt wordt.
public Vector **geefBlokkenVector**()	Geeft de blokkenvector van de te bewerken methode.
public void **veranderSelectie**(int)	Verandert het geselecteerde blok.
public int **geefSelectie**()	Geeft de index van het geselecteerde blok.

Klasse PSDWerkbalk

Dit is het kleine werkbalkje dat onder het PSD wordt weergegeven. Hij wordt grotendeels overgenomen van de PSD Creator-klasse Werkbalk, maar de knoppen voor nieuwe blokken staan er niet meer in, omdat nieuwe blokken in de Applet Creator toegevoegd worden door de tabs aan de zijkant.
Ik heb ook nog een nieuwe knop gemaakt, namelijk Blok bewerken. Uiteindelijk moet deze knop de eigenschappen van het geselecteerde blok aanpassen.
PSDWerkbalk is niet beweegbaar; dat betekent dat de gebruiker de werkbalk niet naar een andere plaats in het scherm kan slepen. Hiervoor zorgt `setFloatable(false)`. Er wordt, net als bij hoofdMenubalk, ook een ActieAfhandelaar aangemaakt die ervoor zorgt dat er iets gebeurt als er op een van de knoppen geklikt wordt.

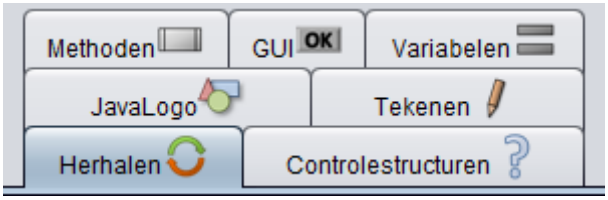
Figuur 20 – Schermafdruk van de PSDWerkbalk.

Ledenoverzicht **PSDWerkbalk**	
Variabelen / Velden	
ActieAfhandelaar **actie**	ActieAfhandelaar die gebruikt wordt om de acties af te handelen die plaatsvinden op deze werkbalk.
Constructoren	
public **PSDWerkbalk**(Methode)	Maakt de werkbalk en zet de knoppen erin.

Methoden	
protected JButton **knopMaken**(String, String, String, String)	Maakt een knop met de opgegeven parameters. Met deze methode worden de werkbalkknoppen gemaakt. Er wordt ook een ActionCommand toegevoegd aan de knop; hiermee weet ActieAfhandelaar welke actie er moet worden uitgevoerd bij het klikken op de knop.

Klasse Blokkentabs

Dit is ook weer een JPanel. Dit JPanel bevat een JTabbedPane, met de gewenste tabbladen. Voor de tabbladen gebruik ik losse JPanels. Het grootste deel van deze klasse bestaat uit de opsomming van alle BlokkenInTabs in de constructor. Van ieder blok dat in de tabs moet komen te staan, wordt namelijk de constructor van BlokInTabs gebruikt om het blok te maken. Daarbij moeten alle variabelen worden opgegeven die bij het blok horen. Deze klasse zorgt er dus onder meer voor dat de blokken "weten" welke Java-code ze genereren als ze worden uitgevoerd.
Ook voor de Blokkentabs heb ik iconen gemaakt. Iedere tab heeft namelijk een klein icoontje om de tab gemakkelijk te kunnen herkennen. Zie *Figuur 21*.



Figuur 21 – Schermafdruk van de blokkentabs.

Ledenoverzicht **Blokkentabs**	
Constructoren	
public **Blokkentabs**()	Maakt een nieuwe Blokkentabs en zet alle blokken erin.
Methoden	
protected static ImageIcon **maakIcoon**(String)	Maakt een icoon aan de hand van de opgegeven naam.

Klasse BlokInTabs

Ook dit is een JPanel. Dit is dus een verschil met een "gewoon" blok; bij een gewoon blok wordt *Graphics.drawRect* gebruikt om te tekenen en bij BlokInTabs wordt gewoon de rechthoekige vorm van het JPanel gebruikt. Ik heb besloten dat ieder BlokInTabs een simpele rechthoek wordt, dus ook keuze- en herhalingsblokken, om het eenvoudiger te maken. Later of in een volgende versie zal ik dit misschien nog veranderen.
Om de gegevens vast te houden, worden dezelfde zes Strings gebruikt als bij een gewoon Blok (zie aldaar). Daarom heb ik ook veel code kunnen kopiëren vanuit Blok. Er is ook nog een andere String; deze bevat de naam van het blok, die wordt weergegeven in de tabs (*blokNaam*). *waardenVanEigenschappenArray* bestaat juist niet in BlokInTabs, want deze wordt niet gebruikt. (De waarden van een blok worden immers pas ingevuld als het blok toegevoegd wordt aan het PSD.)

Ledenoverzicht **BlokInTabs**	
Variabelen / Velden	
public String **blokNaam**	De naam die op het blok staat, als deze in de zijbalk staat.
public String **blokVorm**	De vorm van het blok.
public String **label**	De tekst die op het blok staat.

public String **javaCode**	De code die gegenereerd gaat worden, als het blok omgezet gaat worden naar Java-code.
public String **javaCodeGUI**	De code die in de variabeledeclaratie van de GUI gegenereerd gaat worden.
public String **javaCodeAction**	De code die in de action-methode gegenereerd gaat worden.
public String[] **eigenschappenArray**	Array met beschrijvingen van de door de gebruiker instelbare eigenschappen van het blok.
Constructoren	
public **Blok**(String, String, String, String, String, String, String, Vector)	Maakt een nieuw BlokInTabs. In de Vector moeten de eigenschappen van het blok komen.
public **Blok**(String, String, String, String, String, String, String, String[])	Een extra constructor die de vorige gewoon aanroept, maar eerst een Vector maakt van de array. Dit om te voorkomen dat ik in blokkentabs steeds allerlei Vectoren aan moet maken; arrays zijn dan makkelijker.
Methoden	
public void **paintComponent**(Graphics)	Wordt aangeroepen als het blok getekend wordt: tekent de tekst in het blok.
public void **mousePressed**(MouseEvent) *(en de drie volgende)*	Hoort bij MouseListener, maar wordt niet gebruikt.
public void **mouseClicked**(MouseEvent)	Hoort bij MouseListener. Deze methode wordt aangeroepen als de gebruiker op het blok klikt. Deze methode zorgt er dus voor dat het blok toegevoegd wordt aan het PSD.
Blok **hiervanEenBlokMaken**()	Maakt een Blok van deze BlokInTabs. Wordt gebruikt door mouseClicked.

Klasse ActieAfhandelaar

De klasse die zorgt voor het afhandelen van de events die komen vanuit de menubalk en de PSD-werkbalk. De klasse bestaat eigenlijk slechts uit één methode: *actionPerformed*. Deze methode onderzoekt wat het ActionCommand is van de actie die uitgevoerd moet worden. Dit gebeurt met een serie if-constructies, zoals deze:

```
        if (cmd.equals(AFSLUITEN))
        {
            description = "Het programma afsluiten.";
            System.exit(0);
        }
```

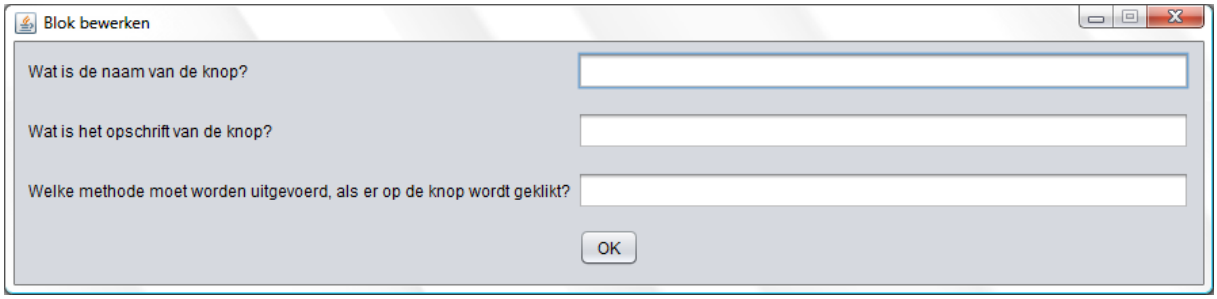
description is daarbij een String die slechts bedoeld is voor de programmeur. Deze kan namelijk naar System.out worden geschreven, zodat duidelijk is waarop de gebruiker heeft geklikt.

Ledenoverzicht **ActieAfhandelaar**	
Methoden	
public void **actionPerformed**(ActionEvent)	Wordt aangeroepen als er een gebeurtenis plaats heeft gevonden. Deze methode zorgt voor het afhandelen van alle acties.

Klasse BlokBewerkenVenster

Deze klasse is een JFrame met daarin invoervelden, zodat het blok kan worden aangepast. De constructor van BlokBewerkenVenster wordt aangeroepen met een Blok. Dit blok is het blok dat bewerkt zal moeten worden met het BlokBewerkenVenster. Dit blok wordt op de variabele *teBewerkenBlok* gezet. Daarna wordt het venster als het ware dynamisch opgebouwd: voor ieder element in de eigenschappenVector van het blok wordt er een *EigenschappenWaardeKoppel* gemaakt en in de *eigenschappenWaardeKoppelsVector* gezet. *EigenschappenWaardeKoppel* is een kleine klasse, het is een JPanel met daarop een JLabel en een JTextField.

Als er dan op OK geklikt wordt, dan zorgt de methode *actionPerformed* ervoor dat alle waarden uit de *EigenschappenWaardeKoppels* worden gelezen en weer in de *waardenVanEigenschappenVector* van *teBewerkenBlok* worden gezet.



Figuur 22 – Voorbeeld van een BlokBewerkenVenster. Dit venster hoort bij het blok Maak een knop (Button).

Ledenoverzicht **BlokBewerkenVenster**	
Constructoren	
public **BlokBewerkenVenster**(Blok)	Maakt een BlokBewerkenVenster. Per eigenschap komt een EigenschappenWaardeKoppel in de *eigenschappenWaardeKoppelsVector* te staan.
Methoden	
public void **actionPerformed**(ActionEvent)	Wordt aangeroepen als er op de OK-knop geklikt wordt. Stelt de waarden van het blok in op de waarden die ingevuld zijn in de EigenschappenWaardeKoppels.
Klassen	
De klasse EigenschappenWaardeKoppel is een JPanel met daarop een JLabel en een JTextField. Iedere eigenschap krijgt zo'n EigenschappenWaardeKoppel.	

Ledenoverzicht **EigenschappenWaardeKoppel**	
Constructoren	
public **EigenschappenWaardeKoppel**(String, String)	Maakt een EigenschappenWaardeKoppel. Deze constructor vraagt twee Strings: de naam van de eigenschap en de waarde waar deze eigenschap op dit moment op is ingesteld.
Methoden	
public String **geefWaarde**()	Geeft de op dit moment ingevulde waarde.

Klasse MethodeBewerkenVenster

Het venster dat voor het bewerken van een door de gebruiker gemaakte methode zorgt. (Initialiseren en Tekenmethode worden bewerkt door StandaardMethodeBewerkenVenster.) In dit venster kunnen worden ingesteld:

- De naam van de methode.
- Of de methode public of private is.
- Welke parameters de methode vraagt en van welke soort die zijn.
- Welke variabelen de methode gebruikt en van welke soort die zijn.
- Wat voor soort resultaat de methode teruggeeft.

De constructor van MethodeBewerkenVenster krijgt een AndereMethode mee bij de aanroep. Deze methode moet er bewerkt worden, en hij wordt in *teBewerkenMethode* gezet. Ook krijgt hij een integer *positieInVector*. De bedoeling daarvan is dat de klasse weet welke methode er

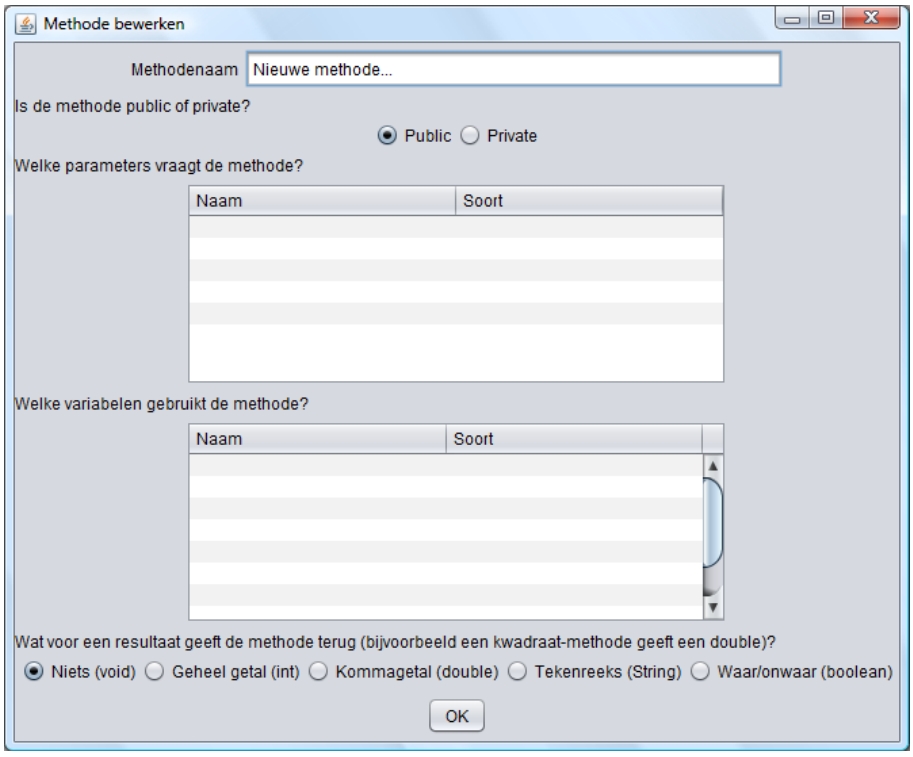
bewerkt wordt. Feitelijk is dat niet van belang, want de klasse kan gewoon *teBewerkenMethode* aanpassen, maar om de Methodenlijst aan de zijkant aan te passen is de positie wel nodig.
De constructor laat het werk van het bouwen van de interface over aan de lange methode *contentPaneMaken*. Deze methode maakt van boven naar beneden het hele contentpane van MethodeBewerkenVenster. Als eerste wordt er een nieuw JPanel aangemaakt met een BoxLayout. Hierna zorgt de methode ervoor dat de volgende elementen worden neergezet in dit JPanel:

- Een JPanel *methodeNaamPanel* met daarin een JLabel en een JTextField, waarin de naam van de methode kan worden ingevuld. In dit vakje wordt ook direct de huidige naam van de methode ingevuld.
- Een JLabel met "Is de methode public of private?".
- Een JPanel *publicPrivatePanel* met daarin twee keuzerondjes.
- Een JLabel met "Welke parameters vraagt de methode?".
- Een JPanel *parameterPanel*. Hierin wordt de JScrollPane met de *parametertabel* geplaatst. Als de gebruiker op een cel in de kolom Soort klikt, dan opent er een JComboBox met de mogelijke keuzes. Dit heb ik gedaan met:

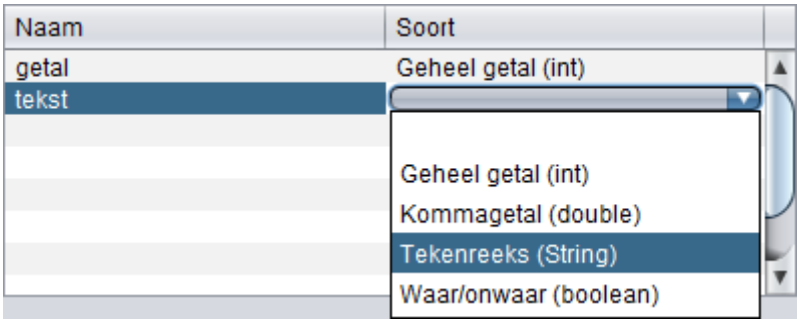
```
parametertabel.getColumnModel().getColumn(1).setCellEditor
                                    (new DefaultCellEditor(comboBox));
```

  Hierin is *comboBox* de JComboBox met waarden, die ik al eerder voorbereid had.
- Een JLabel met "Welke variabelen gebruikt de methode?".
- Een JPanel *varsPanel* met de *varstabel* op dezelfde manier als de *parametertabel*.
- Een JLabel met "Wat voor resultaat geeft de methode terug (bijvoorbeeld een kwadraat-methode geeft een double)?".
- Een JPanel *teruggavePanel*. Hierop staan de vijf keuzerondjes.
- Een JPanel *knopPanel* met daarop de OK-knop.

De methode *actionPerformed* zorgt ervoor dat de opties ook echt worden aangepast in de methode en sluit het venster af.



Figuur 23 – Een voorbeeld van een MethodeBewerkenVenster.

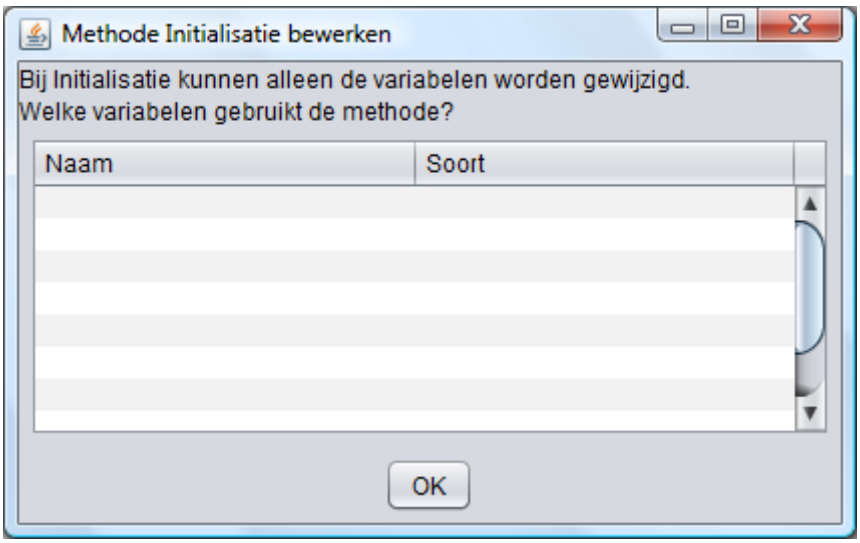


Figuur 24 – Als de gebruiker op een cel in de kolom Soort van de parameter- of de variabelentabel klikt, dan verschijnt er een keuzemenu met daarin alle mogelijke keuzes voor het soort parameter / variabele.

Ledenoverzicht **MethodeBewerkenVenster**	
Constructoren	
public **MethodeBewerkenVenster**(AndereMethode, int)	Maakt een MethodeBewerkenVenster. Gebruikt de methode *contentPaneMaken*.
Methoden	
JPanel **contentPaneMaken**()	Maakt het contentpane van het MethodeBewerkenVenster en zet alle elementen erin.
public void **actionPerformed**(ActionEvent)	Wordt aangeroepen als er op de OK-knop geklikt wordt. Zorgt ervoor dat alle eigenschappen ook echt aangepast worden in de methode.

Klasse StandaardMethodeBewerkenVenster

Het venster dat voor het bewerken van de methoden Initialiseren en Tekenmethode zorgt. Dit is eigenlijk dezelfde klasse als MethodeBewerkenVenster, alleen heb ik alle invoervelden weggehaald, behalve voor de variabelen. Bij Initialiseren en Tekenmethode zijn namelijk alleen de variabelen te wijzigen; de rest van de instelopties ligt vast, omdat die door Java gedefinieerd worden. Tekenmethode heeft bijvoorbeeld altijd een Graphics-parameter.
Verder is *teBewerkenMethode* hier natuurlijk geen AndereMethode, maar een InitialisatieMethode of TekenMethode.



Figuur 25 – Schermafdruk van het StandaardMethodeBewerkenVenster van Initialiseren.

Ledenoverzicht **StandaardMethodeBewerkenVenster**	
Constructoren	
public **StandaardMethodeBewerkenVenster**(Methode, int)	Maakt een StandaardMethodeBewerkenVenster. Gebruikt de methode *contentPaneMaken*.
Methoden	
JPanel **contentPaneMaken**()	Maakt het contentpane van het StandaardMethodeBewerkenVenster en zet alle elementen erin.
public void **actionPerformed**(ActionEvent)	Wordt aangeroepen als er op de OK-knop geklikt wordt. Zorgt ervoor dat alle eigenschappen ook echt aangepast worden in de methode.

Klasse Aboutbox

Een heel simpel venster met drie JLabels erin. Het voornaamste doel van de aboutbox is het weergeven van het versienummer (op dit moment is dat v1.0.0).

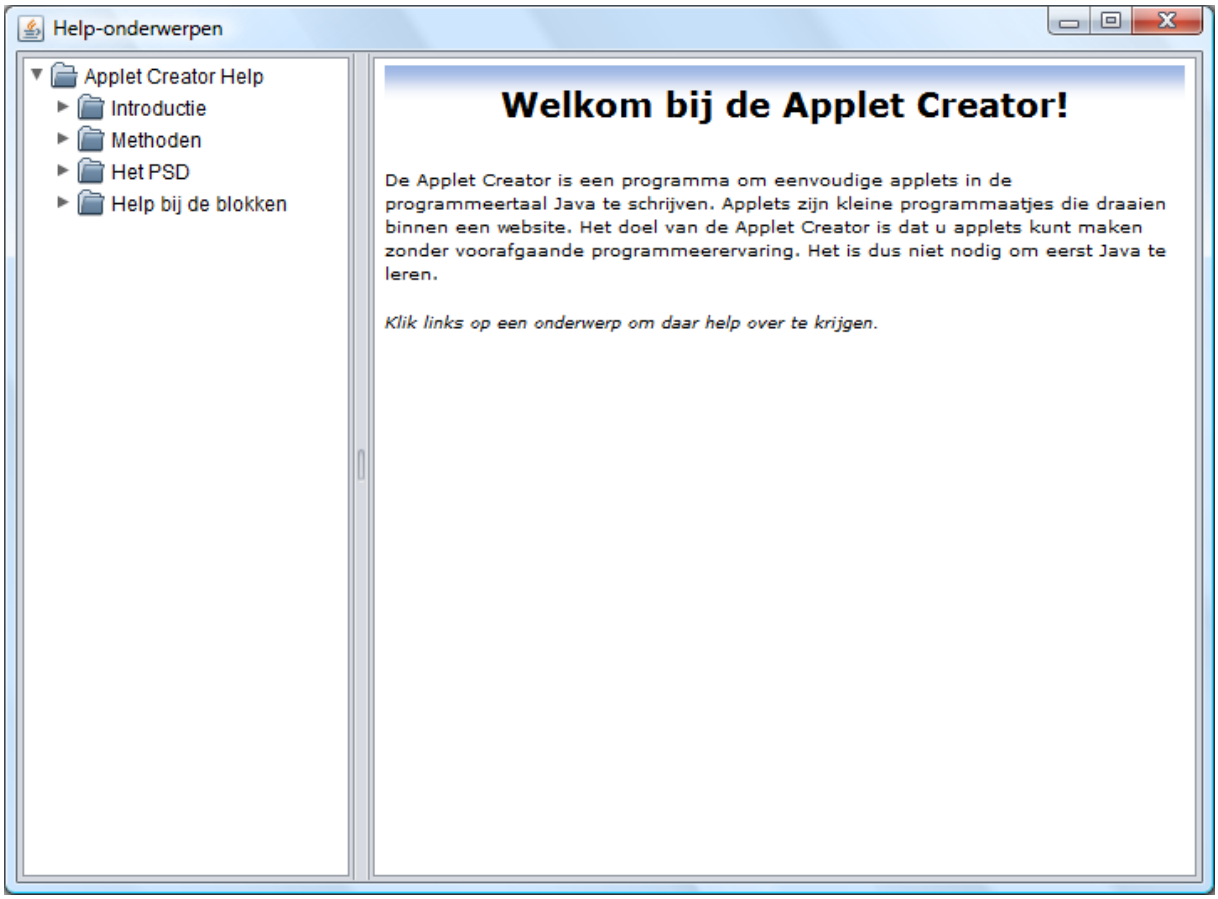
Ledenoverzicht **Aboutbox**	
Constructoren	
public **Aboutbox**()	Maakt de aboutbox en zet de JLabels erin.

Klasse Helpvenster

Helpvenster is een JFrame dat de help-onderwerpen van het programma toont. Deze klasse heb ik met toestemming grotendeels overgenomen uit Ockellaravity[4].
Aan de linkerkant van het scherm staat een lijst met helponderwerpen. Deze onderwerpen zijn gerangschikt in een JTree: ze zijn verdeeld in categorieën. Deze categorieën zijn in- en uitklapbaar door op het pijltje ervoor te klikken.
Als er op een helponderwerp geklikt wordt, dan wordt dit geopend in het venster aan de rechterkant. De helpbestanden zijn in feite HTML-bestanden, die in de submap *help* staan. Ze worden ingelezen door een JEditorPane, die ingebouwde ondersteuning heeft voor HTML. Helaas is deze ondersteuning niet optimaal en heb ik een eenvoudige layout moeten maken. Ik wilde eerst aan H1 een afbeelding koppelen; dit werkte in een browser goed, maar in de JEditorPane liep dit toch mis.
In de helpbestanden staan soms ook hyperlinks; dit werkte eerst niet: de JEditorPane bleek bij het klikken op een hyperlink niet over te springen naar een andere pagina. Dit is opgelost door zelf een HyperlinkListener toe te voegen die handmatig overspringt naar het bestand waarnaar de hyperlink verwijst.

De helpbestanden zelf ga ik overigens pas maken in stap 6 (Documenteren), maar ik heb nu alvast enkele bestanden gemaakt om Helpvenster mee te testen. Zie *Figuur 26* voor een schermafdruk van Helpvenster.

[4] Ockellaravity is een project dat ik samen met Stijn Köster en Laurey-Anne Krammer gemaakt heb bij het vak informatica. Zie ook de Bronnenlijst.



Figuur 26 – Een schermafdruk van het helpvenster met daarin de tekst die wordt weergegeven bij het starten van de Help.

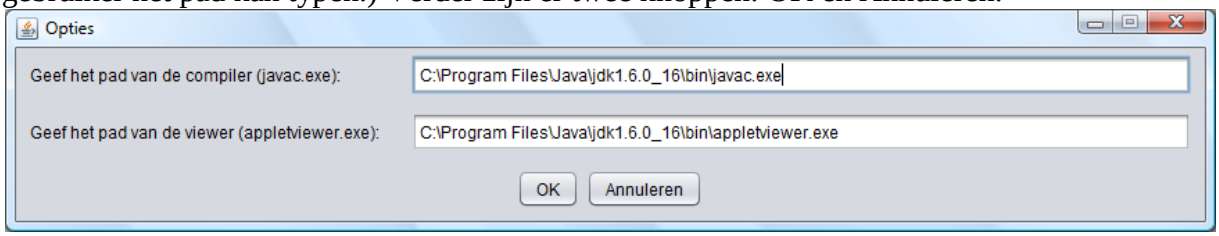
Ledenoverzicht **Helpvenster**	
Variabelen / Velden	
private JEditorPane **htmlPane**	Het stuk van het scherm waarop de helppagina wordt getoond.
private JTree **tree**	Het stuk van het scherm waarop de boom met helponderwerpen wordt getoond.
Constructoren	
public **Helpvenster**()	Maakt het Helpvenster. De twee elementen van het venster worden gemaakt en met een aanroep van de methode *nodesAanmaken* wordt de hele boom van onderwerpen opgebouwd.
Methoden	
public void **valueChanged**(TreeSelectionEvent)	Wordt aangeroepen als er een ander helponderwerp geselecteerd wordt.
public void **hyperlinkUpdate**(HyperlinkEvent)	Wordt aangeroepen als er een event met betrekking tot een hyperlink plaatsvindt. Raar genoeg gebeurt dit ook als er met de muis over de link bewogen wordt. Om te voorkomen dat er dan wordt overgesprongen naar een andere pagina, wordt hierop gecontroleerd met een if-loop.
private void **displayURL**(URL)	Zet de opgegeven URL in de JEditorPane.
private void **nodesAanmaken**(DefaultMutableTreeNode)	Deze methode neemt de top-node van de JTree aan en hangt daar alle helponderwerpen aan. Bij ieder helponderwerp wordt een nieuw HelpOnderwerp-object gemaakt.
Klassen	
De klasse **HelpOnderwerp** bevat een String die de naam van het helponderwerp weergeeft, en een	

URL die het pad naar het bestand weergeeft.

Ledenoverzicht **HelpOnderwerp**	
Constructoren	
public **HelpOnderwerp**(String, String)	Maakt een HelpOnderwerp met de opgegeven naam en de opgegeven bestandsnaam (die naar een URL wordt geconverteerd).
Methoden	
public String **toString**()	Geeft de naam van het helponderwerp.

Klasse OptiesVenster

Dit is het venster waarin de gebruiker de opties van het programma kan instellen; op dit moment is dat alleen het pad van *javac.exe* en *appletviewer.exe*. Hiervoor worden er twee instanties van *Invoer* gemaakt (*Invoer* bestaat uit een JLabel en een JTextField, waarin de gebruiker het pad kan typen.) Verder zijn er twee knoppen: OK en Annuleren.



Figuur 27 – Schermafdruk van het optiesvenster.

Ledenoverzicht **OptiesVenster**	
Constructoren	
public **OptiesVenster**()	Maakt het optiesvenster en zet de invoervakken erin.
Methoden	
public void **actionPerformed**(ActionEvent)	Wordt aangeroepen als er op OK of op Annuleren geklikt wordt.
Klassen	
De klasse **Invoer** bevat een JLabel, die een beschrijving van de in te voeren waarde bevat, en een JTextField, waarin de gebruiker de waarde kan typen. Er zijn methoden om de waarde op te halen of te wijzigen.	

Ledenoverzicht **Invoer**	
Constructoren	
Invoer (String)	Maakt een Invoer met de opgegeven tekst erbij.
Methoden	
String **geefWaarde**()	Geeft de waarde die ingevuld is in het JTextField.
void **veranderWaarde**(String)	Verandert de waarde in het JTextField.

De opslag van het PSD maken

Klasse MethodeVerzameling

MethodeVerzameling is een klasse die alle methoden bijeenhoudt. De klasse bevat een Vector, de methodenVector, waarin alle methoden staan. Standaard staan hierin al de InitialiserenMethode en de TekenMethode. Er zijn ook methoden om een methode toe te voegen of te verwijderen.

Ledenoverzicht **MethodeVerzameling**	
Variabelen / Velden	
Vector<Methode> **methodenVector**	De Vector waarin alle methoden staan.
Constructoren	
public **MethodeVerzameling**()	Maakt een nieuwe MethodeVerzameling. Standaard worden alvast de Initialisatie- en de Tekenmethode ingeladen in de *methodenVector*.
Methoden	
public AndereMethode **methodeToevoegen**(String)	Voegt een nieuwe AndereMethode toe aan de *methodenVector* met de opgegeven naam.
public void **methodeVerwijderen**(int)	Verwijdert de methode op de opgegeven index. Als index ≤ 1, dan wordt er niets verwijderd (want Initialisatie en Tekenmethode kunnen niet verwijderd worden).
public Methode **geefMethode**(int)	Geeft de methode op de opgegeven index.
public int **geefAantalMethoden**()	Geeft het aantal methoden in de *methodenVector*.

Interface Methode

Alle klassen die een methode voorstellen in het programma (namelijk InitialisatieMethode, TekenMethode en AndereMethode) voldoen aan deze interface. Het nut hiervan is duidelijk als we bijvoorbeeld kijken naar de methode veranderTeTekenenPSD in PSDPaneel.

```
    public void veranderTeTekenenPSD(Methode nieuweMethode)
    {
        teTekenenBlokkenVector = nieuweMethode.geefBlokkenVector();
        repaint();
    }
```

Deze methode vraagt een Methode, dus eigenlijk vraagt hij om “een klasse die de interface Methode implementeert”. Aangezien alle implementerende klassen geefBlokkenVector() moeten bevatten, kan deze methode aangeroepen worden, onafhankelijk van welke soort Methode er eigenlijk gegeven is. De interface Methode bevat dus methodedefinities die voor iedere soort methode gelden.

Ledenoverzicht **Methode** (interface)	
Methoden	
public Vector **geefBlokkenVector**()	Geeft de blokkenVector van het PSD van deze methode terug.
public void **veranderNaam**(String)	Verandert de methodenaam.
public String **geefNaam**()	Geeft de methodenaam.
public void **veranderPublic**(boolean)	Verandert of de methode public of private is.
public boolean **isPublic**()	Geeft een boolean terug of de methode public is of niet. Als de boolean *false* is, dan is de methode private.
public void **veranderParameterNamen**(Vector<String>)	Verandert de parameternamen.
public Vector **geefParameterNamen**()	Geeft de parameternamen.
public void **veranderParameterSoorten**(Vector<String>)	Verandert de parametersoorten.
public Vector **geefParameterSoorten**()	Geeft de parametersoorten.
public void **veranderVariabeleNamen**(Vector<String>)	Verandert de variabelenamen.
public Vector **geefVariabeleNamen**()	Geeft de variabelenamen.
public void **veranderVariabeleSoorten**(Vector<String>)	Verandert de variabelesoorten.
public Vector **geefVariabeleSoorten**()	Geeft de variabelesoorten.
public void **veranderReturnType**(String)	Verandert het returntype. Het returntype wordt weergegeven als een String.
public string **geefReturnType**()	Geeft het returntype. Het returntype wordt weergegeven als een String.

Klasse InitialisatieMethode

Implementeert Methode. Deze klasse geeft de methode Initialisatie weer.

Ledenoverzicht **InitialisatieMethode**	
Variabelen / Velden	
Vector **blokkenVector**	De blokkenVector van Initialisatie.
Constructoren	
public **InitialisatieMethode**()	Maakt een InitialisatieMethode met lege variabelenamen en –soorten.
Methoden	
De methoden van interface *Methode*.	

Klasse TekenMethode

Implementeert Methode. Deze klasse geeft de methode Tekenmethode weer.

Ledenoverzicht **TekenMethode**	
Variabelen / Velden	
Vector **blokkenVector**	De blokkenVector van de Tekenmethode.
Constructoren	
public **TekenMethode**()	Maakt een TekenMethode met lege variabelenamen en –soorten.
Methoden	
De methoden van interface *Methode*.	

Klasse AndereMethode

Implementeert Methode. Deze klasse geeft een door de gebruiker aangemaakte methode weer. In tegenstelling tot InitialisatieMethode en TekenMethode, kunnen bij AndereMethode ook de naam, de parameters en het returntype van de methode worden aangepast.

Ledenoverzicht **AndereMethode**	
Variabelen / Velden	
Vector **blokkenVector**	De blokkenVector van de methode.
Constructoren	
public **AndereMethode**()	Maakt een AndereMethode met standaardeigenschappen.
Methoden	
De methoden van interface *Methode*.	

Klasse Blok

Een simpele klasse die de eigenschappen van een blok opslaat. De klasse bestaat uit zeven Strings die te wijzigen zijn.

Ledenoverzicht **Blok**	
Variabelen / Velden	
public String **blokVorm**	De vorm van het blok.
public String **label**	De tekst die op het blok staat.
public String **javaCode**	De code die gegenereerd gaat worden, als het blok omgezet gaat worden naar Java-code.
public String **javaCodeGUI**	De code die in de variabeledeclaratie van de GUI gegenereerd gaat worden.
public String **javaCodeAction**	De code die in de action-methode gegenereerd gaat worden.
public Vector **eigenschappenVector**	Vector met beschrijvingen van de door de gebruiker instelbare eigenschappen van het blok.
public Vector **waardenVanEigenschappenVector**	Vector met de waarden die voor de instelbare eigenschappen door de gebruiker ingegeven zijn.

Constructoren	
public **Blok**(String, String, String, String, String, Vector, Vector)	Maakt een nieuw Blok met de opgegeven waarden.
Methoden	
public static String **vervangEigenschappen**(String, Vector)	Vervangt alle in te vullen eigenschappen in de String door de overeenkomstige waarden in de Vector.

De uitvoer

De uitvoer van de Java-code wordt verzorgd door de klasse UitvoerModule. Om het gemaakte programma ook direct uit te voeren, is het ook nodig dat er een HTML-pagina gemaakt wordt en dat de gemaakte code gegenereerd wordt. De klasse HTMLUitvoer zorgt voor de HTML-pagina; het compileren gebeurt in ActieAfhandelaar.

Klasse UitvoerModule

UitvoerModule heeft slechts één *public* methode, namelijk *voerJavaCodeUit*. Deze methode voert de applet uit naar het opgegeven bestand. *voerJavaCodeUit* gebruikt de andere methoden in de klasse:

1. *genereerGUIVariabeleDeclaratie()* – maakt de variabelendeclaratie van de GUI-elementen
2. *genereerInitCode()* – genereert de body van de methode *init*
3. *genereerPaintCode()* – genereert de body van de methode *paint*
4. *genereerActionCode()* – genereert de body van de methode *action*
5. *genereerCodeAndereMethoden()* – genereert de code van de andere methoden, inclusief de methodeheaders
6. *schrijf(String)* – schrijft de meegegeven String naar het bestand, houdt daarbij rekening met de huidige inspringing
7. *schrijfVector(Vector)* – schrijft alle elementen in de Vector naar het bestand

voerJavaCodeUit gebruikt een BufferedWriter om naar het bestand te kunnen schrijven. Deze BufferedWriter wordt ingesteld op ASCII, om te voorkomen dat de compiler problemen krijgt met de (UTF-8). Van de BufferedWriter wordt de methode *write* gebruikt om een String naar het bestand te schrijven.
Hieronder staat een voorbeeld van een (nutteloze) applet die door deze klasse gegenereerd is, met erachter een getal dat aangeeft welke methode welke code gemaakt heeft. De getallen komen overeen met de lijst hierboven.

```
// Gegenereerd door Java Applet Creator v1.0.0

import java.awt.*;
import java.applet.*;

public class TestKlasse extends Applet
{
    Button knopje;} 1

    public void init()
    {
        String nuttelozeTekst;                    }
        knopje = new Button("Knop"); add(knopje);} 2
    }

    public void paint(Graphics pen)
    {
        pen.drawString("Hallo", 20, 20);} 3
    }

    public boolean action(Event e, Object o)
```

```
    {
        if (e.target == knopje) { this.klikken(); return true; }
        return false;
    }

    public void klikken()
    {

    }

}
```

4

5

Ledenoverzicht **UitvoerModule**	
Variabelen / Velden	
BufferedWriter **out**	De BufferedWriter waarmee naar de uitvoer geschreven wordt.
Methoden	
public void **voerJavaCodeUit**(String)	Voert de Java-applet uit naar het bestand met de opgegeven bestandsnaam.
void **schrijf**(String)	Schrijft de meegegeven String naar het bestand.
void **schrijfVector**(Vector)	Schrijft alle elementen van de opgegeven Vector naar het bestand.
Vector **genereerInitCode**()	Genereert de body van de init-methode.
Vector **genereerGUIVariabeleDeclaratie**()	Genereert de declaratie van de GUI-elementen.
Vector **genereerPaintCode**()	Genereert de body van de paint-methode.
Vector **genereerActionCode**()	Genereert de body van de action-methode.
Vector **genereerCodeAndereMethoden**()	Genereert de code van de andere methoden.
String **omzettenVariabeleType**(String)	Zet de meegegeven String (bijvoorbeeld “Geheel getal (int)” om naar een door Java bruikbaar equivalent (bijvoorbeeld “int”).

Klasse HTMLModule

HTMLModule zorgt voor het maken van een HTML-pagina waarin de applet staat. De code is voor een deel gebaseerd op UitvoerModule, maar hij is hier veel simpeler. De “hoofdmethode” heet hier *voerHTMLUit*, maar hij werkt ongeveer hetzelfde als bij UitvoerModule. *voerHTMLUit* doet trouwens alles zelf (behalve het schrijven naar het bestand, waarvoor *schrijf(String)* gebruikt wordt).

Een voorbeeld van een door HTMLModule gemaakt HTML-document (dat hoort bij de Testklasse hiervoor):

```
<html>
<body bgcolor=cyan>
<applet code="TestKlasse.class" width=500 height=500>Uw browser ondersteunt
geen applets.</applet>
</body>
</html>
```

De achtergrondkleur van de pagina is blauw, om te kunnen zien waar de rand van de applet is.

Ledenoverzicht **HTMLModule**	
Variabelen / Velden	
BufferedWriter **out**	De BufferedWriter waarmee naar de uitvoer geschreven wordt.
Methoden	
public void **voerHTMLUit**(String)	Voert de HTML-pagina uit voor de Java-code met de opgegeven bestandsnaam.
void **schrijf**(String)	Schrijft de meegegeven String naar het bestand.

Aanpassingen aan ActieAfhandelaar

ActieAfhandelaar moet aangepast worden, zodat het uitvoeren ook gestart wordt bij het klikken op een menuoptie.

- Bij het klikken op **Java-code genereren…** wordt de volgende code uitgevoerd:

```
            JFileChooser fc = new JFileChooser();
            int returnVal = fc.showSaveDialog(null);

            if (returnVal == JFileChooser.APPROVE_OPTION) {
                File file = fc.getSelectedFile();
                UitvoerModule uitvoer = new UitvoerModule();
                try
                {
                    HoofdContentPane.statusbalk.veranderStatusTekst
                                          ("Code wordt gegenereerd.");
                    uitvoer.voerJavaCodeUit(file.getCanonicalPath());
                    HoofdContentPane.statusbalk.veranderStatusTekst
                                                                 ("");
                }
                catch (IOException fout)
                {
                    JOptionPane.showMessageDialog(null,
                        "Er is iets misgegaan bij het uitvoeren. "+
                        "Probeer een andere bestandsnaam./n/n"+
                        "Foutcode: 1.",
                        "Applet Creator",
                         JOptionPane.ERROR_MESSAGE);
                }
            }
```

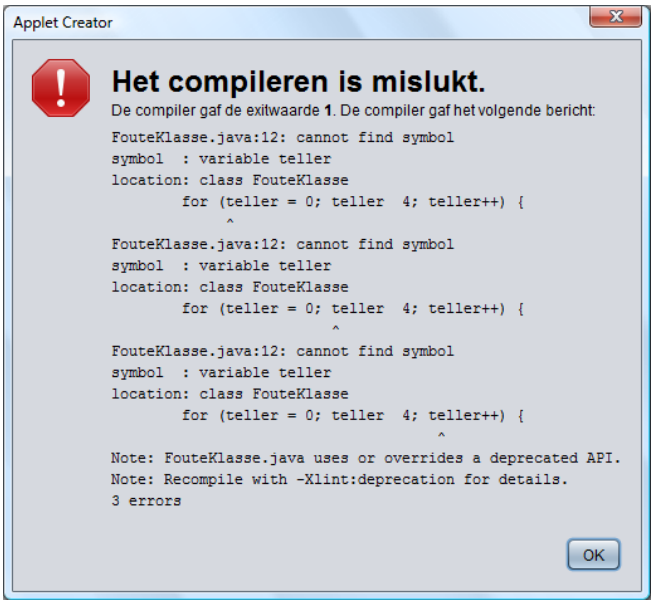
Als eerste wordt er een opslaan-dialoogvenster geopend waarin de gebruiker aan kan geven waar de code moet worden neergezet. Hierna wordt er gecontroleerd of er in dit venster op OK geklikt is. Als dat het geval is, dan wordt UitvoerModule gebruikt om de code te genereren met behulp van het pad uit het dialoogje. Mocht er een IOException worden opgegooid, dan wordt die afgevangen en wordt er een foutmelding gegeven.

- Bij het klikken op **Java-code genereren en uitvoeren…** wordt de volgende code uitgevoerd (commentaar en onbelangrijke regels weggelaten, met cijfers ter referentie):

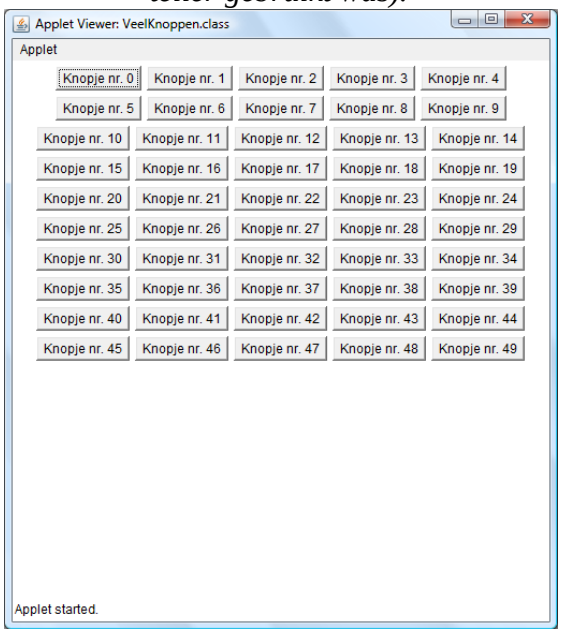
```
            JFileChooser fc = new JFileChooser(); ❶
            int returnVal = fc.showSaveDialog(null);

            if (returnVal == JFileChooser.APPROVE_OPTION) { ❷
                File file = fc.getSelectedFile();
                UitvoerModule uitvoer = new UitvoerModule();
                try
                {
                    uitvoer.voerJavaCodeUit
                                        (file.getCanonicalPath()); ❸
                    ProcessBuilder pb = new ProcessBuilder
                          (AppletCreator.javacPad, file.getName()); ❹
                    pb.directory(file.getParentFile());
                    Process javac = pb.start();

                    BufferedReader javacStream = new BufferedReader
                      (new InputStreamReader(javac.getErrorStream()));

                    String s = null;
                    String foutmelding = "";
                    while ((s = javacStream.readLine()) != null) ❺
                    {
                        foutmelding = foutmelding + s + "<br>";
                    }

                    int exitwaarde = javac.waitFor(); ❻
```

```
                    javacStream.close();

                    if (exitwaarde != 0) ❼
                    {
                        // FOUTMELDING WEERGEVEN
                    }
                    else
                    {
                        // HTML-pagina maken:
                        HTMLModule html = new HTMLModule();
                        html.voerHTMLUit(file.getCanonicalPath()); ❽

                        // De Appletviewer openen:
                        String bestandsnaamZonderExtensie = file.
                          getCanonicalPath().replaceAll(".java", "");
                        ProcessBuilder pb2 = new ProcessBuilder
    (AppletCreator.viewerPad, bestandsnaamZonderExtensie + ".htm"); ❾
                        pb.directory(file.getParentFile());
                        System.out.println("1");
                        Process appviewer = pb2.start();

                        BufferedReader appStream = new BufferedReader
                  (new InputStreamReader(appviewer.getErrorStream()));
                        while ((s = appStream.readLine()) != null)
                        {
                            System.out.println("AppViewer: "+s);
                        }

                        int exitwaarde2 = appviewer.waitFor(); ❿
                        appStream.close();
                    }
                }
                // CATCH IOException en InterruptedException
            }
```

Deze code is ingewikkelder, omdat ook *javac* en *appletviewer* gestart moeten worden. Eerst wordt weer het opslaan-venster getoond. (❶). Als er daarin op OK is gedrukt (❷), dan gebeurt er het volgende. Als eerste wordt de Java-code gegenereerd (❸). Hierna wordt met behulp van een ProcessBuilder *javac* aangeroepen (❹). Het pad hiervoor wordt gehaald uit de klasse AppletCreator. Met de methode *ProcessBuilder.start* wordt *javac* gestart. Hierna wordt de errorstream van *javac* uitgelezen om de eventuele foutmeldingen op te vangen; bovendien wordt hiermee voorkomen dat *javac* crasht (❺). Er wordt gewacht tot *javac* klaar is en de exitwaarde wordt opgevangen (❻). Bij *javac* geldt dat de compilatie goed is verlopen als de foutwaarde 0 is; is de foutwaarde 1 dan is er een fout opgetreden. Er wordt dus gecontroleerd of de foutwaarde 0 is (❼). Als dat zo is, dan wordt de eerder gelezen errorstream in een foutmelding aan de gebruiker gegeven. Als de foutwaarde wel 0 is, en de compilatie dus goed is gegaan, dan wordt er een HTML-pagina gemaakt met de HTMLModule (❽) en daarna wordt deze met de *appletviewer* geopend, weer met een ProcessBuilder (❾). Ook van *appletviewer* wordt de errorstream gelezen en er wordt weer gewacht tot het programma klaar is (❿). Hierna is de methode afgelopen, dus dan keert de uitvoering van het programma weer terug naar de GUI.



Figuur 28 – De foutmelding die de gebruiker krijgt als er een fout in de gegenereerde broncode zat (de variabele teller *was niet gedeclareerd, terwijl die in een for-loop als teller gebruikt was).*



Figuur 29 – Een voorbeeld van een applet in de appletviewer. De applet is door de Applet Creator gecompileerd en weergegeven, met de hierboven beschreven code.

Openen en opslaan

Ik heb besloten dat de openen- en opslaanfuncties wegens tijdgebrek nog niet worden geïmplementeerd in deze versie van de Applet Creator. Dit is wel jammer, want dit zorgt ervoor dat gemaakte projecten niet kunnen worden opgeslagen om er later aan verder te gaan. Misschien ga ik dit in een volgende versie nog inbouwen. Als de gebruiker op Openen of Opslaan klikt, dan wordt er een foutmelding gegeven.

Stap 4 is nu klaar.

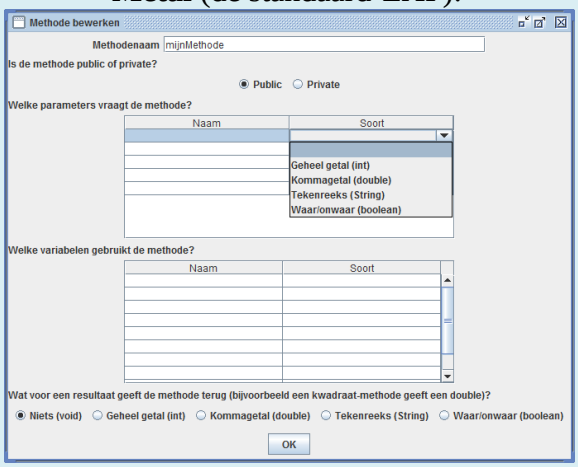
Stap 5: Testen

Omdat de doelstelling van het project was dat de Applet Creator alle programma's uit het boekje *Java: my cup of tea* moest kunnen maken, ga ik als test van alle programma's uit dit boekje een Applet Creator-versie maken.

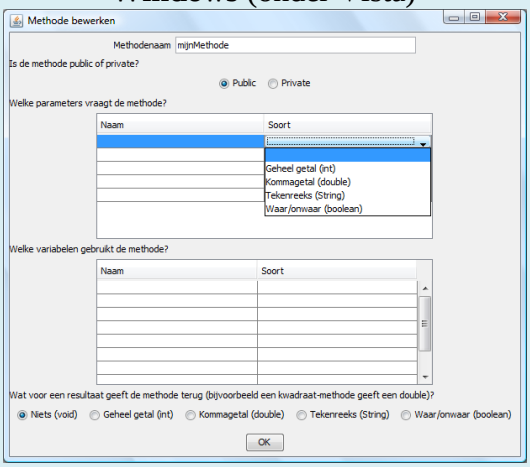
Look-and-feels

Swing ondersteunt verschillende "lay-outs" voor de besturingselementen, de zogenaamde look-and-feels (LAF's). Tot nu toe heb ik altijd gewerkt met Nimbus. Hieronder staat een schermafdruk van het MethodeBewerkenVenster met verschillende LAF's:

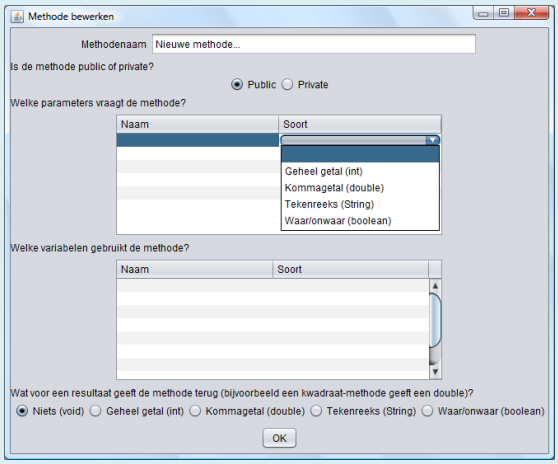
Metal (de standaard-LAF):



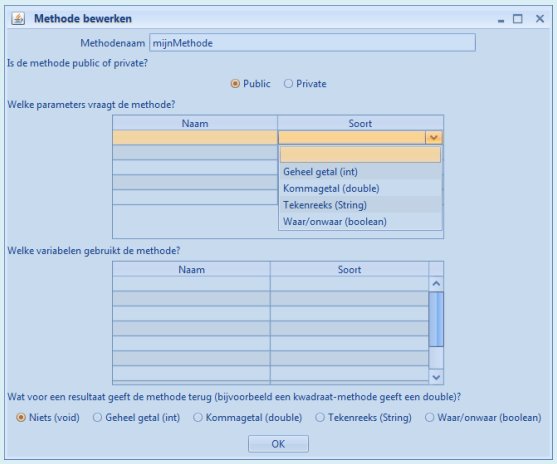
Windows (onder Vista)



Nimbus



Substance *



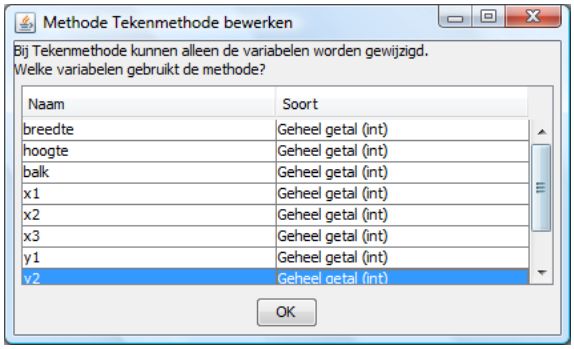
* Substance is een LAF die niet standaard bij Java hoort.

Zoals te zien is, werkt het venster in alle LAF's goed. Voortaan zal ik afwisselend verschillende LAF's gebruiken bij het testen, want misschien gebruiken bepaalde gebruikers andere LAF's en dan moet het programma wel goed blijven werken.

De Mondriaan-applet

Het programma Mondriaan in het boekje *Java: my cup of tea* is een applet die een statische tekening weergeeft in de stijl van Mondriaan. De tekening wordt opgebouwd uit rechthoekige vlakken, waarvan vijf zwarte balken en twee gekleurde vlakken (één rood en één blauw vlak).

Het enige wat ik dus moet doen in de Applet Creator is de Tekenmethode vullen met de tekenopdrachten.
Als eerste vul ik de variabelen in (zie *Figuur 30*). Dit ging goed; er waren geen problemen.
Daarna heb ik de variabelen waarden gegeven met het blok *Stel een variabele in*. Daarbij was er een fout: als je op OK klikt in het BlokBewerkenVenster, dan wordt het blok niet direct bijgewerkt.
Als laatste heb ik de rechthoeken laten tekenen. Mijn uiteindelijke PSD zag er zo uit:



Figuur 30 – De variabelen van de Tekenmethode van de Mondriaan-applet.

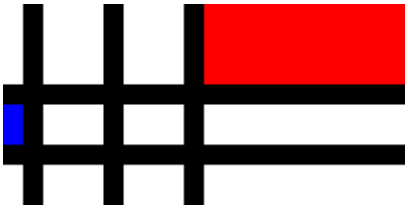
Stel variabele breedte in op 200
Stel variabele x1 in op 10
Stel variabele x2 in op 50
Stel variabele x3 in op 90
Stel variabele hoogte in op 100
Stel variabele y1 in op 40
Stel variabele y2 in op 70
Stel variabele balk in op 10
Verander de tekenkleur in white
Teken gevulde rechthoek van (0, 0) tot (breedte, hoogte)
Verander de tekenkleur in black
Teken gevulde rechthoek van (x1, 0) tot (balk, hoogte)
Teken gevulde rechthoek van (x2, 0) tot (balk, hoogte)
Teken gevulde rechthoek van (x3, 0) tot (balk, hoogte)
Teken gevulde rechthoek van (0, y1) tot (breedte, balk)
Teken gevulde rechthoek van (0, y2) tot (breedte, balk)
Verander de tekenkleur in blue
Teken gevulde rechthoek van (0, y1 + balk) tot (x1, y2 - (y1+balk))
Verander de tekenkleur in red
Teken gevulde rechthoek van (x3 + balk, 0) tot (breedte - (x3+balk), y1)

Ik heb de code laten genereren en laten opslaan als *Mondri.java*. Het resultaat was:

```
// Gegenereerd door Java Applet Creator v1.0.0

import java.awt.*;
import java.applet.*;

public class Mondri extends Applet
{

    public void init()
    {
    }

    public void paint(Graphics pen)
    {
        int breedte;
        int hoogte;
        int balk;
        int x1;
        int x2;
```

```
        int x3;
        int y1;
        int y2;
        breedte = 200;
        x1 = 10;
        x2 = 50;
        x3 = 90;
        hoogte = 100;
        y1 = 40;
        y2 = 70;
        balk = 10;
        pen.setColor(Color.white);
        pen.fillRect(0, 0, breedte, hoogte);
        pen.setColor(Color.black);
        pen.fillRect(x1, 0, balk, hoogte);
        pen.fillRect(x2, 0, balk, hoogte);
        pen.fillRect(x3, 0, balk, hoogte);
        pen.fillRect(0, y1, breedte, balk);
        pen.fillRect(0, y2, breedte, balk);
        pen.setColor(Color.blue);
        pen.fillRect(0, y1 + balk, x1, y2 - (y1+balk));
        pen.setColor(Color.red);
        pen.fillRect(x3 + balk, 0, breedte - (x3+balk), y1);
    }

    public boolean action(Event e, Object o)
    {
        return false;
    }

}
```

Deze code zag er goed uit, dus hierna heb ik de code ook laten compileren en uitvoeren door de Applet Creator. Het resultaat is te zien in *Figuur 31*.



Figuur 31 – Het resultaat van de Mondriaan-applet.

De applet ziet er net zo uit als de voorbeeldafbeelding uit het boekje.

Probleem: blok wordt niet ververst bij sluiten BlokBewerkenVenster

Het enige probleem dat ik dus naar aanleiding van deze test op moet lossen, is het bijwerken van de blokken als het BlokBewerkenVenster gesloten wordt. Dit heb ik opgelost door in BlokBewerkenVenster (in de methode *actionPerformed*) het PSD te laten verversen:

```
        HoofdContentPane.psdGebied.psdPaneel.repaint();
```

Huisjes en methoden

De volgende applet gebruikt een aparte methode, naast de *paint*-methode, om een huisje te tekenen. Hierna wordt deze methode aangeroepen vanuit *paint*. Om deze applet te maken in de Applet Creator, moet ik dus een nieuwe methode maken.
Als eerste heb ik weer de variabelen van de Tekenmethode gedeclareerd (*x* en *y*) en ze een waarde gegeven. De oplossing voor het vorige probleem werkt trouwens goed. Bij het aanroepen van de zelfgemaakte methode, is er echter een probleem.

Probleem: het Graphics-object in eigen methoden krijgen

De code uit het boekje luidt:

```
        this.tekenHuis(pen, x, y, 40);
```

Hoe ga ik nu het Graphics-object *pen* meegeven? Dit is een heel lastig probleem. Immers: de eigen methoden kunnen geen Graphics-object ontvangen! Ik zou natuurlijk iedere methode automatisch een Graphics-object kunnen laten ontvangen en dan bij een aanroep automatisch dit object meegeven, maar dan zijn ze weer niet aan te roepen vanuit Initialiseren, want daar is geen Graphics-object beschikbaar.
De gebruiker moet dus zelf kunnen kiezen of de methode een Graphics-object vraagt of niet. Dit heeft consequenties:

- als er géén Graphics-object gevraagd wordt, dan kan er niet getekend worden binnen de methode;
- als er wel een Graphics-object gevraagd wordt, dan kan er wel getekend worden, maar de methode kan dan slechts worden aangeroepen vanuit een methode die al zo'n object bezit, dus feitelijk alleen vanuit de Tekenmethode.

Dit kan ik natuurlijk wel inbouwen, maar dan is er nog een probleem. Hoe ga ik nu bepalen bij het uitvoeren van de Java-code hoe ik wil dat de methode wordt aangeroepen? Immers, gewoon de variabele javaCode van het aanroepblok wordt gekopieerd, daarbij is er geen if-statement te maken dat controleert of de methode een Graphics nodig heeft of niet. Dit alles is dus heel ingewikkeld en het zou zeker een enorme verandering in de code betekenen.

Toen bedacht ik een radicaal andere oplossing. Het schema in *Figuur 32* hieronder komt uit de API-documentatie van het Java-platform. Dit betekent dat Applet *extends* Panel, dat Panel *extends* Container, enzovoorts.

Figuur 32 – Het schema met de superklassen van Applet.

Applet "is" dus een Component en Component heeft een methode *getGraphics()*. Deze methode geeft het Graphics-object van het Component-object. Ik kan nu aan het begin van iedere methode die door de gebruiker gemaakt is een aanroep naar *getGraphics()* zetten, waardoor het Graphics-object toch als *pen* beschikbaar komt. De gebruiker merkt daar niets van en kan gewoon zonder problemen blokken in de methode zetten.
Voor dit inderdaad te doen heb ik onderzocht of dit wel werkt. Ik wist immers niet of het uitmaakt of je een nieuw Graphics-object maakt of dat je het oude houdt. Ik heb daarvoor de klasse Huizen uit het boekje omgeschreven naar gebruik van *getGraphics()*:

```
import java.applet.*;
import java.awt.*;

public class Huizen extends Applet
{
    private void tekenHuis(int x, int y, int breed)
    {
        Graphics pen = getGraphics();  // Graphics-object verkrijgen
        int topx, topy;
        topx = x + breed/2;
        topy = y - 3*breed / 2;
        pen.drawRect(x, y-breed, breed, breed);
        pen.drawLine(x, y-breed, topx, topy);
        pen.drawLine(topx, topy, x+breed, y-breed);
```

```
    }

    public void paint (Graphics pen)
    {
        int x = 20; int y = 100;
        pen.drawLine(0, 0, 200, 200);
        this.tekenHuis(x, y, 40); x+=50;
        this.tekenHuis(x, y, 40); x+=50;
        this.tekenHuis(x, y, 60);
    }
}
```

Dit blijkt inderdaad net zo te werken als het programma in het boekje. Ik ga er nu dus voor zorgen dat iedere door de gebruiker gemaakte methode de volgende regel als eerste regel krijgt:

```
Graphics pen = getGraphics();
```

Dit doe ik door in de UitvoerModule in de methode *genereerCodeAndereMethoden()* na de regels die de methodeheader te maken deze regel toe te voegen:

```
            v.add("    Graphics pen = getGraphics();");
```

Dit is natuurlijk niet zo netjes als er toevallig geen tekenblok in deze methode staat, maar ik weet geen andere eenvoudige oplossing, behalve om ieder blok in de methode te gaan controleren om te zien of er een tekenblok instaat. Dit lijkt me te ingewikkeld om in te bouwen.

Nu ga ik weer verder met het maken van het huizen-programma in de Applet Creator. Als eerste heb ik de Tekenmethode gemaakt; hierna de methode *tekenHuis*.
Mijn PSD's zijn:

Tekenmethode:

Stel variabele x in op 20
Stel variabele y in op 100
Voer methode tekenHuis uit
Stel variabele x in op x + 50
Voer methode tekenHuis uit
Stel variabele x in op x + 50
Voer methode tekenHuis uit

tekenHuis:

Stel variabele topx in op x + breed/2
Stel variabele topy in op y - 3*breed/2
Verander de tekenkleur in red
Teken gevulde rechthoek van (x, y - breed) tot (breed, breed)
Verander de tekenkleur in black
Teken rechthoek van (x, y - breed) tot (breed, breed)
Teken lijn van (x, y - breed) tot (topx, topy)
Teken lijn van (topx, topy) tot (x + breed, y - breed)

Daarna heb ik de code laten genereren en laten opslaan als Huizen2.java. Het resultaat:

```
// Gegenereerd door Java Applet Creator v1.0.0

import java.awt.*;
import java.applet.*;

public class Huizen2 extends Applet
{
```

```
    public void init()
    {
    }

    public void paint(Graphics pen)
    {
        int x;
        int y;
        x = 20;
        y = 100;
        this.tekenHuis(x, y, 40);
        x = x + 50;
        this.tekenHuis(x, y, 40);
        x = x + 50;
        this.tekenHuis(x, y, 60);
    }

    public boolean action(Event e, Object o)
    {
        return false;
    }

    private void tekenHuis(Geheel getal (int) x, Geheel getal (int) y,
Geheel getal (int) breed, )
    {
        Graphics pen = getGraphics();
        int topx;
        int topy;

        topx = x + breed/2;
        topy = y - 3*breed/2;
        pen.setColor(Color.red);
        pen.fillRect(x, y - breed, breed, breed);
        pen.setColor(Color.black);
        pen.drawRect(x, y - breed, breed, breed);
        pen.drawLine(x, y - breed, topx, topy);
    }

}
```

De toevoeging van de nieuwe regel werkt, maar er gaat ergens iets helemaal mis bij de methodeheader van *tekenHuis*.

Probleem: fouten bij methodenheaders van eigen methoden

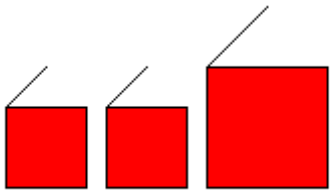
Op de een of andere manier worden de parametertypen niet omgezet. Dat bleek bij een nadere bestudering van de broncode heel logisch: ik had de aanroep van *omzettenVariabeleType* vergeten. Deze aanroep heb ik dus toegevoegd.
Er is echter nog een probleem met die methodeheader: er staat een komma achter de laatste parameter, wat niet zou moeten. Om dit te voorkomen, had ik het volgende stukje code ingebouwd:

```
                if (teller2 != parNamen.size())
                {
                    parameters = parameters + ", ";
                }
```

teller2 is daarbij een teller die van *0* tot *parNamen.size()-1* loopt. Dat is juist het probleem: *teller2* zal altijd ongelijk zijn aan *parNamen.size()*. Dit stukje code moet dus worden vervangen door:

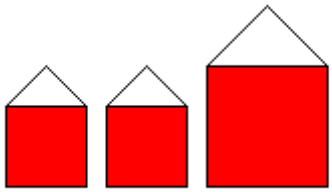
```
                if (teller2 != parNamen.size() - 1)
                {
                    parameters = parameters + ", ";
                }
```

Hierna heb ik de code opnieuw laten genereren en compileren. Het resultaat staat hieronder in *Figuur 33*.



Figuur 33 – De Huizen-applet.

Dat gaat blijkbaar toch nog niet helemaal goed; één helft van het dak ontbreekt. Dat is echter niet de fout van de Applet Creator: ik had de andere lijn gewoon vergeten. Deze lijn heb ik dus toegevoegd en het resultaat is nu wel hoe het bedoeld was:



Figuur 34 – De Huizen-applet verbeterd.

Applet met een GUI

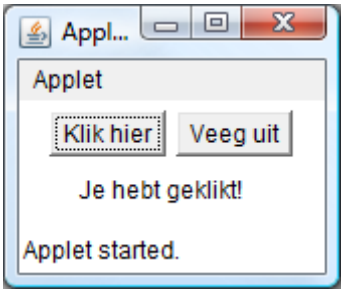
Deze applet heeft twee knopjes: “Klik hier” en “Veeg uit”. Als er op “Klik hier” wordt geklikt, dan verschijnt de tekst “Je hebt geklikt!” in een Label. Als er daarna op “Veeg uit” wordt geklikt, dan verdwijnt deze tekst weer. In de Applet Creator moet ik dus de Initialiseren-methode vullen en twee aparte methoden maken die worden aangeroepen als er een knopklik plaatsvindt.
Als eerste heb ik Initialiseren gemaakt. Het maken van een GUI gaat erg makkelijk met de Applet Creator als je het vergelijkt met “echt” Java, waarbij je eerst moet declareren, daarna initialiseren en daarna toevoegen.
Hierna heb ik twee kleine methoden *vullen* en *wissen* gemaakt. *vullen* zorgt er ook voor dat het tekstvak de juiste grootte krijgt.

De gegenereerde code is: (VullenWissen.java):

```
// Gegenereerd door Java Applet Creator v1.0.0

import java.awt.*;
import java.applet.*;

public class VullenWissen extends Applet
{
    Button knopje1;
    Button knopje2;
    Label tekst;

    public void init()
    {
        knopje1 = new Button("Klik hier"); add(knopje1);
        knopje2 = new Button("Veeg uit"); add(knopje2);
        tekst = new Label(""); add(tekst);
    }

    public void paint(Graphics pen)
    {
```

```
    }

    public boolean action(Event e, Object o)
    {
        if (e.target == knopje1) { this.vullen(); return true; }
        if (e.target == knopje2) { this.wissen(); return true; }
        return false;
    }

    public void vullen()
    {
        Graphics pen = getGraphics();

        tekst.setText("Je hebt geklikt!");
        tekst.setBounds(160, 80, 100, 12);
    }

    public void wissen()
    {
        Graphics pen = getGraphics();

        tekst.setText("");
    }

}
```

Compileren en uitvoeren geeft het resultaat van *Figuur 35*.



Figuur 35 – Het resultaat van de VullenWissen-applet.

Ook dit resultaat is in orde.

De Reken-applet

Van deze applet staat in het boekje alleen wat de applet moet kunnen, er staat geen code bij. Dit is dus een goede manier om te testen of je ook zelfstandig programma's kunt maken met de Applet Creator.
De bedoeling van de Reken-applet is dat er twee invoervakken staan, waarin de gebruiker getallen kan typen, en drie knoppen voor optellen, aftrekken en vermenigvuldigen. Voor deze applet is het dus van belang om een getal te kunnen lezen uit een invoervak. Dit bestaat uit twee losse problemen:

- de String uit het vakje lezen
- de String omzetten naar een int waarmee gerekend kan worden

Dit kan al met de Applet Creator, maar dit is voor mensen die niet gevorderd zijn in Java bijna niet te doen, zie *Figuur 36*.

Figuur 36 – Op dit moment is het veel te ingewikkeld om de tekst uit een invoervakje te halen en dit om te zetten naar een getal.

Uiteraard is dit niet te begrijpen voor mensen die niet bekend zijn met Java. Daarom maak ik twee nieuwe blokken onder de tab Variabelen:

- **Verander een tekenreeks in een geheel getal**
  Opties die gegeven kunnen worden:
  - Welke tekenreeks moet geconverteerd worden? (*teConverteren*)
  - In welke variabele moet het gehele getal gezet worden? (*variabele*)

  De gegenereerde code is:

```
variabele = Integer.parseInt(teConverteren);
```

- **Verander een tekenreeks in een kommagetal**
  Opties die gegeven kunnen worden:
  - Welke tekenreeks moet geconverteerd worden? (*teConverteren*)
  - In welke variabele moet het kommagetal gezet worden? (*variabele*)

  De gegenereerde code is:

```
variabele = Double.parseDouble(teConverteren);
```

Verder komt er nog een blok onder de tab GUI:

- **Zet de inhoud van een invoervak in een variabele**
  Opties die gegeven kunnen worden:
  - Van welk invoervak? (*invoerVak*)
  - In welke (tekenreeks-)variabele moet de inhoud van het invoervak gezet worden? (*variabele*)

  De gegenereerde code is:

```
variabele = invoervak.getText();
```

Met deze nieuwe blokken is het eenvoudiger om de tekst uit een invoervak uit te lezen en er een getal van te maken. Het kan nu bijvoorbeeld met het volgende PSD, waarbij de hulpvariabele *hulp* gebruikt wordt:

Zet inhoud van tekstvakje in hulp
Verander hulp in geheel getal resultaat

Verder komt er nog een blok onder de tab GUI om de standaard-layoutmanager te verwijderen, om te voorkomen dat die met de afmetingen gaat knoeien:

- **Verwijder de standaardlayout**
  Opties die gegeven kunnen worden:
  - *geen*

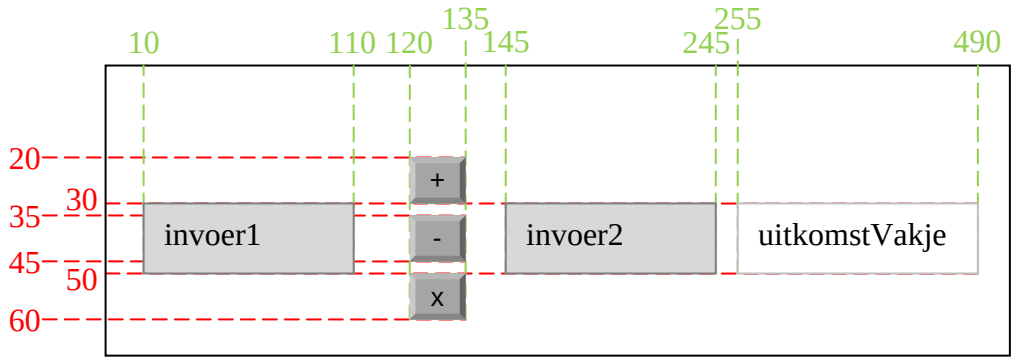
  De gegenereerde code is:

```
setLayout(null);
```

Bij het testen van dit blok bleek echter, dat er dan een leeg BlokBewerkenVenster verschijnt met alleen een OK-knop erin. Dit is natuurlijk verwarrend. Daarom heb ik BlokBewerkenVenster aangepast:

```
        if (aantalEigenschappen == 0)
        {
            contentpane.add(new JLabel("<html><i>Geen eigenschappen om te
                                                    bewerken</i></html"));
        }
```

Nu ga ik de applet maken. Als eerste gebruik ik het nieuwe blok *Verwijder de standaardlayout*. Hierna maak ik de invoervakken en de knoppen met de volgende coördinaten:



Figuur 37 – Schematische weergave van de elementen van de Reken-applet.

Mijn PSD van Initialiseren is als volgt:

Verwijder de standaardlayout
Maak het invoervak invoer1
Zet invoer1 op (10, 30) met formaat 100x20
Maak de knop plusKnop
Zet plusKnop op (120, 20) met formaat 15x10
Maak de knop minKnop
Zet minKnop op (120, 35) met formaat 15x10
Maak de knop keerKnop
Zet keerKnop op (120, 50) met formaat 15x10
Maak het invoervak invoer2
Zet invoer2 op (145, 30) met formaat 100x20
Maak het vaste tekstvak uitkomstVakje
Zet uitkomstVakje op (255, 30) met formaat 235x20

Hierna moet ik de methoden *optellen*, *aftrekken* en *vermenigvuldigen* maken. Mijn PSD van *optellen* is als volgt:

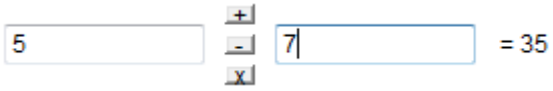
Zet inhoud van invoer1 in hulp
Verander hulp in geheel getal getal1
Zet inhoud van invoer2 in hulp
Verander hulp in geheel getal getal2
Stel variabele uitkomst in op "= " + (getal1 + getal2)
Tekst van uitkomstVakje veranderen in uitkomst

De PSD's van *aftrekken* en *vermenigvuldigen* heb ik op dezelfde manier gemaakt.
Hierna heb ik de code laten genereren (Reken.java):

```
// Gegenereerd door Java Applet Creator v1.0.0

import java.awt.*;
import java.applet.*;

public class Reken extends Applet
{
    TextField invoer1;
    Button plusKnop;
    Button minKnop;
    Button keerKnop;
    TextField invoer2;
    Label uitkomstVakje;

    public void init()
    {
        setLayout(null);
        invoer1 = new TextField(10); add(invoer1);
        invoer1.setBounds(10, 30, 100, 20);
        plusKnop = new Button("+"); add(plusKnop);
        plusKnop.setBounds(120, 20, 15, 10);
        minKnop = new Button("-"); add(minKnop);
        minKnop.setBounds(120, 35, 15, 10);
        keerKnop = new Button("x"); add(keerKnop);
        keerKnop.setBounds(120, 50, 15, 10);
        invoer2 = new TextField(10); add(invoer2);
        invoer2.setBounds(145, 30, 100, 20);
        uitkomstVakje = new Label(""); add(uitkomstVakje);
        uitkomstVakje.setBounds(255, 30, 235, 20);
    }

    public void paint(Graphics pen)
    {
    }

    public boolean action(Event e, Object o)
    {
        if (e.target == plusKnop) { this.optellen(); return true; }
        if (e.target == minKnop) { this.aftrekken(); return true; }
        if (e.target == keerKnop) { this.vermenigvuldigen(); return true; }
        return false;
    }

    private void optellen()
    {
        Graphics pen = getGraphics();
        String hulp;
        int getal1;
        int getal2;
        String uitkomst;

        hulp = invoer1.getText();
        getal1 = Integer.parseInt(hulp);
        hulp = invoer2.getText();
        getal2 = Integer.parseInt(hulp);
        uitkomst = "= " + (getal1 + getal2);
        uitkomstVakje.setText(uitkomst);
    }

    private void aftrekken()
    {
        Graphics pen = getGraphics();
        String hulp;
        int getal1;
        int getal2;
        String uitkomst;

        hulp = invoer1.getText();
        getal1 = Integer.parseInt(hulp);
```

```
        hulp = invoer2.getText();
        getal2 = Integer.parseInt(hulp);
        uitkomst = "= " + (getal1 - getal2);
        uitkomstVakje.setText(uitkomst);
    }

    private void vermenigvuldigen()
    {
        Graphics pen = getGraphics();
        String hulp;
        int getal1;
        int getal2;
        String uitkomst;

        hulp = invoer1.getText();
        getal1 = Integer.parseInt(hulp);
        hulp = invoer2.getText();
        getal2 = Integer.parseInt(hulp);
        uitkomst = "= " + (getal1 * getal2);
        uitkomstVakje.setText(uitkomst);
    }

}
```

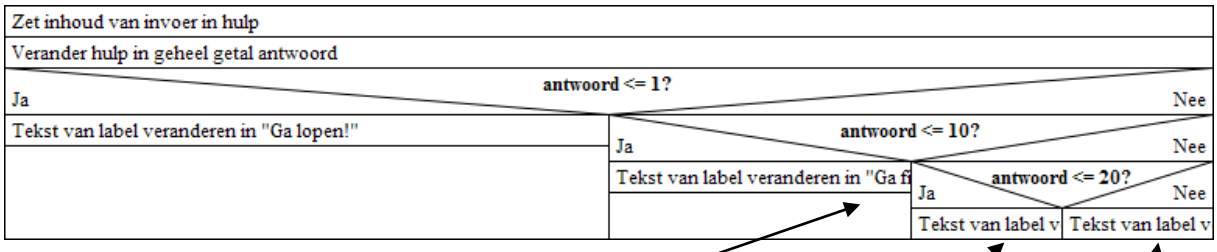
Het is eigenlijk niet zo netjes dat de aanroep in eigen methoden om een Graphics-object te krijgen vóór de variabeledeclaratie staat. Dit heb ik dus veranderd in de code van de Applet Creator. De code van de Reken-applet compileerde toch zonder fouten en gaf de applet van *Figuur 38*.



Figuur 38 – Het resultaat van de Reken-applet.

De Reisadvies-applet

In de Reisadvies-applet moet de gebruiker zijn afstand tot school ingeven. Daarna geeft het programma aan of je moet lopen, fietsen, "bussen" of een school dichter bij huis moet zoeken. Dit is dus een mooie test voor if-constructies in de Applet Creator.
In de Initialiseren-methode heb ik een Label, een TextField en een Button gemaakt. De methode *geklikt* (die wordt uitgevoerd als er op de knop wordt geklikt) ziet er als volgt uit:



"Ga fietsen!"

"Ga bussen!"

"Zoek een school dichter bij huis!"

Helaas is de tekenmethode van het PSDPaneel niet zo slim dat hij de linkerkant van het "Ga lopen!"-blok wat smaller maakt, dus past niet alle tekst erop. Ik heb daarom met pijlen aangegeven welk blok welke tekst weergeeft.

De gegenereerde code ziet er als volgt uit (Reisadvies.java):

```
// Gegenereerd door Java Applet Creator v1.0.0

import java.awt.*;
import java.applet.*;

public class Reisadvies extends Applet
{
    Label label;
    TextField invoer;
    Button knopje;

    public void init()
    {
        label = new Label("Geef de afstand van huis naar school op
                                    (afgerond op gehele km's):"); add(label);
        invoer = new TextField(3); add(invoer);
        knopje = new Button("Bevestig"); add(knopje);
    }

    public void paint(Graphics pen)
    {
    }

    public boolean action(Event e, Object o)
    {
        if (e.target == knopje) { this.geklikt(); return true; }
        return false;
    }

    private void geklikt()
    {
        String hulp;
        int antwoord;

        Graphics pen = getGraphics();

        hulp = invoer.getText();
        antwoord = Integer.parseInt(hulp);
        if (antwoord <= 1) {
        label.setText("Ga lopen!");
        } else {
        if (antwoord <= 10) {
        label.setText("Ga fietsen!");
        } else {
        if (antwoord <= 20) {
        label.setText("Ga bussen!");
        } else {
        label.setText("Zoek een school dichter bij huis!");
        }
        }
        }
    }

}
```

Het is niet zo mooi dat de if-constructies niet inspringen, maar daar is volgens mij geen eenvoudige oplossing voor.

Het resultaat na compileren staat hieronder.

Figuur 39 – Het resultaat van de Reisadvies-applet.

Concentrische cirkels

Deze applet tekent een reeks concentrische cirkels met behulp van waarden voor het aantal cirkels en hun onderlinge afstand die de gebruiker ingeeft. Hiervoor heb ik wel een nieuw blok nodig, namelijk om *repaint()* te kunnen aanroepen.

repaint()
repaint() tekent, zoals de naam al zegt, de applet opnieuw. Normaal gesproken is dit niet nodig, maar als er iets in de applet moet worden veranderd naar aanleiding van een gebeurtenis binnen de applet, dan moet de paint-methode worden aangeroepen. Het is dan beter om *repaint()* aan te roepen in plaats van de paint-methode.

Het blok komt onder de tab Tekenen te staan.

- **Teken de applet opnieuw**
  Opties die gegeven kunnen worden:
  - *geen*

  De gegenereerde code is:

```
repaint();
```

Er is wel een probleem: in de Applet Creator is het niet mogelijk om variabelen voor de hele klasse te maken. Ik kan dus niet doorgeven welke waarden er door de gebruiker ingegeven zijn. Ik volg dus een andere tactiek om het programma te laten werken: als er op de knop wordt geklikt wordt alleen *repaint()* aangeroepen en de paint-methode leest dan de waarden uit de invoervakjes in.
Mijn PSD van Initialiseren was:

Maak het vaste tekstvak label1
Maak het invoervak aantalInvoer
Maak het vaste tekstvak label2
Maak het invoervak afstandInvoer
Maak de knop knop
Tekst in aantalInvoer veranderen in "10"
Tekst in afstandInvoer veranderen in "5"

En van Tekenmethode:

Zet inhoud van aantalInvoer in hulp
Verander hulp in geheel getal aantalCirkels
Zet inhoud van afstandInvoer in hulp
Verander hulp in geheel getal afstand
Herhaal aantalCirkels keer met teller teller
Teken ovaal van (250 - afstand * teller, 250 - afstand * teller) tot (2 * afstand * teller, 2 * afstand * teller)

Ik zie nu overigens dat de tekst op het blok *Teken een ovaal* onjuist is. Nu staat er namelijk:
Teken ovaal van (*xBoven*, *yLinks*) tot (*breedte*, *hoogte*)
en dat moet natuurlijk zijn
Teken ovaal van (*xBoven*, *yLinks*) met afmetingen *breedte*x*hoogte*
Dit heb ik nu aangepast; net zoals bij de rechthoek, gevulde rechthoek en gevulde ovaal.

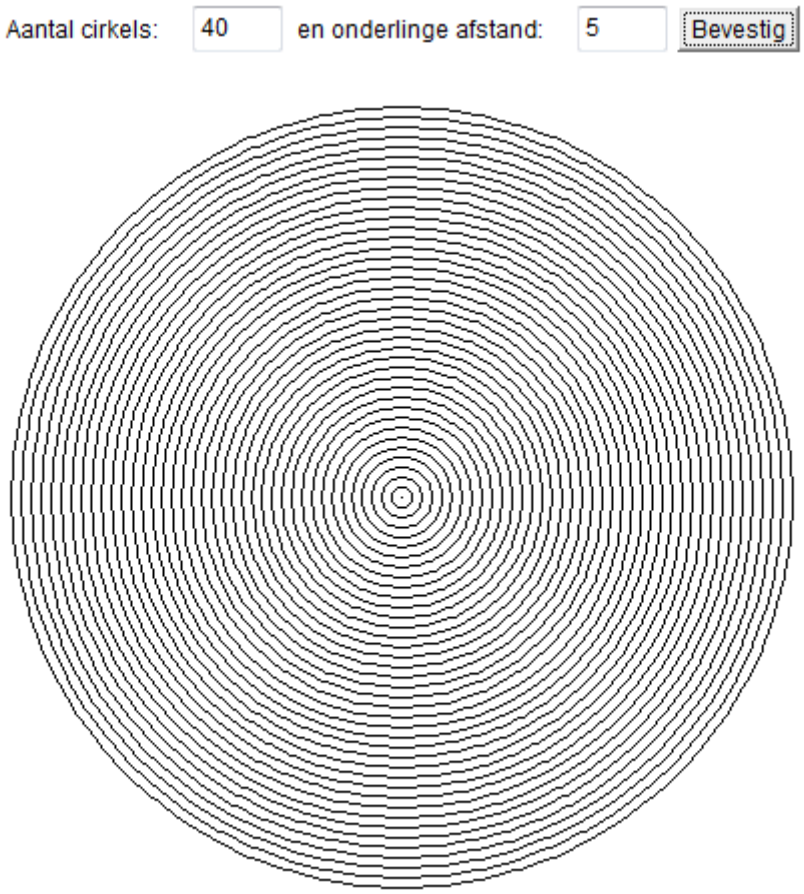
In de methode *klikken* wordt, zoals eerder gezegd, alleen maar het scherm ververst:

Teken de applet opnieuw

De gegenereerde code staat hieronder.

```
// Gegenereerd door Java Applet Creator v1.0.0

import java.awt.*;
import java.applet.*;

public class Concentrisch extends Applet
{
    Label label1;
    TextField aantalInvoer;
    Label label2;
    TextField afstandInvoer;
    Button knop;

    public void init()
    {
        label1 = new Label("Aantal cirkels:"); add(label1);
        aantalInvoer = new TextField(3); add(aantalInvoer);
        label2 = new Label("en onderlinge afstand:"); add(label2);
        afstandInvoer = new TextField(3); add(afstandInvoer);
        knop = new Button("Bevestig"); add(knop);
        aantalInvoer.setText("10");
        afstandInvoer.setText("5");
    }

    public void paint(Graphics pen)
    {
        String hulp;
        int aantalCirkels;
        int afstand;
        int teller;
        hulp = aantalInvoer.getText();
        aantalCirkels = Integer.parseInt(hulp);
        hulp = afstandInvoer.getText();
        afstand = Integer.parseInt(hulp);
        for (teller = 0; teller < aantalCirkels; teller++) {
        pen.drawOval(250 - afstand * teller, 250 - afstand * teller, 2 *
                                    afstand * teller, 2 * afstand * teller);
        }
    }

    public boolean action(Event e, Object o)
    {
        if (e.target == knop) { this.klikken(); return true; }
        return false;
    }

    private void klikken()
    {

        Graphics pen = getGraphics();

        repaint();
    }

}
```

Het resultaat van deze applet staat in *Figuur 40*.



Figuur 40 – Het resultaat van de concentrische-cirkels-applet.

De Priem-applet

De laatste applet in het boekje is de Priem-applet. De bedoeling daarvan is om te bepalen of een getal priem is of niet. (Een priemgetal is een getal met precies twee delers.) In het boekje wordt het volgende PSD gebruikt om te bepalen of een getal priem is of niet:

Zet het ingevoerde getal op teTestenGetal
Kopieer teTestenGetal naar deler
Herhaal totdat deler = 1
Verminder deler met 1
Bepaal teTestenGetal % deler
Is dit nul? Ja / Nee
Zet deler op 1 / ↓
Zet priem op false
Is priem false? Ja / Nee
teTestenGetal is niet priem / teTestenGetal is priem

Dit PSD kan ik gebruiken om het programma in de Applet Creator te zetten. Als eerste ga ik de Initialiseren-methode maken. In deze methode maak ik zoals gebruikelijk de GUI:

Maak het vaste tekstvak label
Maak het invoervak invoerVakje
Maak de knop knopje

In de methode *klikken* wordt getest of het getal priem is aan de hand van het hierboven beschreven PSD.

Stel variabele priem in op true	
Zet inhoud van invoerVakje in hulp	
Verander hulp in geheel getal teTestenGetal	
Stel variabele deler in op teTestenGetal	
Herhaal zolang deler != 1	
Verlaag variabele deler	
Stel variabele restwaarde in op teTestenGetal % deler	
restwaarde == 0? Ja	Nee
Stel variabele deler in op 1	↓
Stel variabele priem in op false	
priem == false? Ja	Nee
Tekst van label veranderen in teTestenGetal + " is niet priem"	Tekst van label veranderen in teTestenGetal + " is priem"

Dit programma werkte echter niet naar behoren: volgens het programma was 1 priem en alle andere getallen niet. Het bleek dat in mijn versie van het algoritme de lus nog uitgevoerd voor deler = 1, want er staat **Herhaal zolang deler != 1**, en daarna werd deler nog verlaagd, dus dan wordt de lus ook nog uitgevoerd voor deler = 1. Daarom dacht het programma dat ieder getal deelbaar is, namelijk door zichzelf. Ik heb deze regel dus vervangen door **Herhaal zolang deler != 2**. Verder heb ik de regel **Stel variabele deler in op 1** vervangen door **Stel variabele deler in op 2**.

De gegenereerde code is (Priem.java):

```
// Gegenereerd door Java Applet Creator v1.0.0

import java.awt.*;
import java.applet.*;

public class Priem extends Applet
{
    Label label;
    TextField invoerVakje;
    Button knopje;

    public void init()
    {
        label = new Label("Geef een positief, geheel getal in.");
                                                                add(label);
        invoerVakje = new TextField(4); add(invoerVakje);
        knopje = new Button("OK"); add(knopje);
    }

    public void paint(Graphics pen)
    {
    }

    public boolean action(Event e, Object o)
    {
        if (e.target == knopje) { this.klikken(); return true; }
        return false;
    }

    private void klikken()
    {
        String hulp;
        int teTestenGetal;
        int deler;
        int restwaarde;
```

```
        boolean priem;

        Graphics pen = getGraphics();

        priem = true;
        hulp = invoerVakje.getText();
        teTestenGetal = Integer.parseInt(hulp);
        deler = teTestenGetal;
        while (deler != 2) {
        deler--;
        restwaarde = teTestenGetal % deler;
        if (restwaarde == 0) {
        deler = 2;
        priem = false;
        } else {
        }
        }
        if (priem == false) {
        label.setText(teTestenGetal + " is niet priem");
        } else {
        label.setText(teTestenGetal + " is priem");
        }
    }

}
```

Het resultaat staat hieronder in *Figuur 41*.

Figuur 41 – Het resultaat van de Priem-applet.

Dit is het einde van stap 5.

Stap 6: Documenteren

In deze laatste stap ga ik meerdere dingen doen:

- de helpbestanden binnen het programma schrijven;
- de handleiding van het programma maken, die gaat over het opstarten van het programma. Als dat gelukt is, dan wordt verwezen naar de helpbestanden in het programma zelf;
- het programma inpakken in een JAR-bestand, om ervoor te zorgen dat gebruikers het gemakkelijk op kunnen starten;
- een webpagina voor het programma maken, zodat de gebruikers het programma kunnen downloaden.

De helpbestanden schrijven

De helpbestanden binnen het programma moeten het voor de gebruiker duidelijk maken hoe het programma werkt; er moet dus ook informatie inzitten over wat PSD's zijn, wat methoden zijn enzovoorts.
Via de klasse Helpvenster kunnen helponderwerpen in HTML met een boom worden weergegeven. Hiervoor heb ik de volgende indeling bedacht. Achter ieder onderwerp staat de naam van het HTML-bestand van dat onderwerp.

- Introductie *(Intro.htm)*
  - Wat is de Applet Creator? *(Intro_WatIs.htm)*
  - Het hoofdscherm *(Intro_Hoofdscherm.htm)*
  - De menubalk *(Intro_Menubalk.htm)*
- Methoden *(Meth.htm)*
  - Wat zijn methoden? *(Meth_WatZijn.htm)*
  - Een methode toevoegen *(Meth_Toevoegen.htm)*
  - Een bestaande methode bewerken *(Meth_Bewerken.htm)*
  - Een bestaande methode verwijderen *(Meth_Verwijderen.htm)*
- Het PSD *(PSD.htm)*
  - Wat is een PSD? *(PSD_WatIs.htm)*
  - De PSD-werkbalk *(PSD_Werkbalk.htm)*
  - Een blok toevoegen *(PSD_BlokToev.htm)*
  - Een blok bewerken *(PSD_BlokBew.htm)*
  - Een blok verplaatsen *(PSD_BlokVerpl.htm)*
  - Een blok verwijderen *(PSD_BlokVerw.htm)*
- Help bij de blokken *(Blok.htm)*
  - Herhalen *(Blok_Herh.htm)*
    - Herhaal een bepaald aantal keer *(Blok_Herh_1.htm)*
    - Herhaal zolang een voorwaarde waar is *(Blok_Herh_2.htm)*
  - Controlestructuren *(Blok_Contr.htm)*
    - Keuze op basis van een vraag *(Blok_Contr_1.htm)*
  - JavaLogo *(Blok_JavaLogo.htm)*
  - Tekenen *(Blok_Teken.htm)*
    - Schrijf tekst op het scherm *(Blok_Teken_1.htm)*
    - Teken een lijn *(Blok_Teken_2.htm)*
    - Teken een rechthoek *(Blok_Teken_3.htm)*
    - Teken een gevulde rechthoek *(Blok_Teken_4.htm)*
    - Teken een ovaal *(Blok_Teken_5.htm)*
    - Teken een gevulde ovaal *(Blok_Teken_6.htm)*

        - Verander de tekenkleur *(Blok_Teken_7.htm)*
        - Teken de applet opnieuw *(Blok_Teken_8.htm)*
    - Methoden *(Blok_Meth.htm)*
        - Voer een methode uit *(Blok_Meth_1.htm)*
        - Een waarde teruggeven *(Blok_Meth_2.htm)*
    - GUI *(Blok_GUI.htm)*
        - Maak een knop (Button) *(Blok_GUI_1.htm)*
        - Maak een invoervak (TextField) *(Blok_GUI_2.htm)*
        - Maak een vast tekstvak (Label) *(Blok_GUI_3.htm)*
        - Stel de positie van een element in *(Blok_GUI_4.htm)*
        - De tekst op een knop aanpassen *(Blok_GUI_5.htm)*
        - De tekst in een invoervak aanpassen *(Blok_GUI_6.htm)*
        - De tekst in een vast tekstvak aanpassen *(Blok_GUI_7.htm)*
        - Zet de inhoud van een invoervak in een variabele *(Blok_GUI_8.htm)*
        - Verwijder de standaardlayout *(Blok_GUI_9.htm)*
    - Variabelen *(Blok_Vars.htm)*
        - Stel een variabele in *(Blok_Vars_1.htm)*
        - Verhoog een variabele met 1 *(Blok_Vars_2.htm)*
        - Verlaag een variabele met 1 *(Blok_Vars_3.htm)*
        - Verander een tekenreeks in een geheel getal *(Blok_Vars_4.htm)*
        - Verander een tekenreeks in een kommagetal *(Blok_Vars_5.htm)*
- Uitvoeren *(Uitv.htm)*
    - Java-code genereren *(Uitv_Genereren.htm)*
    - Het gemaakte programma automatisch starten *(Uitv_Starten.htm)*

Als eerste heb ik het Helpvenster zo aangepast, dat deze pagina's ook weergegeven worden aan de linkerkant. Daarna heb ik de HTML-pagina's geschreven.

Figuur 42 – Schermafdruk van het Helpvenster met een van de helppagina's erin.

De handleiding

De gebruikershandleiding staat hieronder en hij staat ook als PDF-bestand op de CD. Hij is niet erg lang, en bevat informatie over het installeren. Ook staat er een soort stappenplan voor het maken van de eerste applet, dat bedoeld is als een kort overzicht van de mogelijkheden. Voor overige helpinformatie wordt verwezen naar de helpinformatie in het programma en de website.

Handleiding Applet Creator

Inleiding

Met de Applet Creator kunt u met behulp van PSD's een Java-applet maken. Hiervoor is kennis van Java niet nodig; de Applet Creator zorgt zelf voor alle code.

De Applet Creator starten

Om de Applet Creator te starten (onder Windows), dubbelklikt u op het bestand *AppletCreator.jar*. Er hoeft niets geïnstalleerd te worden.
Let wel op: voor de Applet Creator is de JRE van Java nodig. Als het starten van het programma mislukt, ga dan naar **http://www.java.com/** en download de nieuwste versie. Probeer het daarna nog eens.
Als u de gemaakte applets ook wilt testen, dan moet u in plaats van de JRE de JDK downloaden van **http://java.sun.com/javase/downloads/index.jsp**. De JDK zorgt er namelijk voor dat u de gemaakte programma's ook kunt compileren (omzetten naar een door de computer leesbare code); met de JRE is het alleen mogelijk om al gecompileerde programma's te draaien. Let op: als u de JDK heeft gedownload, is het niet meer nodig om ook de JRE te downloaden, want de JRE zit standaard ook bij de JDK.
Onder andere besturingssystemen dan Windows zou het programma ook moeten werken, maar dit is niet getest. Ook weet ik niet of het opstarten daar op dezelfde manier gaat.

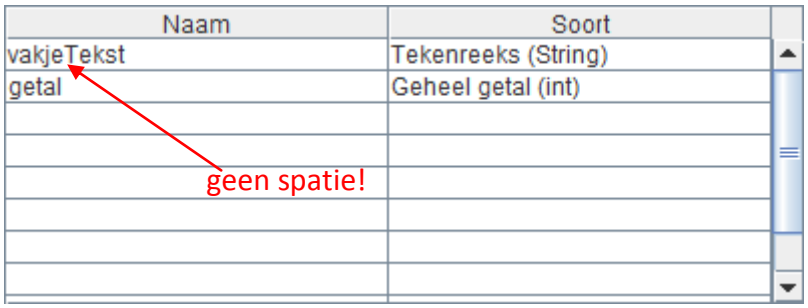
Uw eerste applet

In deze sectie vindt u een kort overzicht van de mogelijkheden van de Applet Creator door een eenvoudige applet te maken. Wilt u meer informatie, kijk dan in de helpinformatie van het programma (*Help > Help* in het menu).
We gaan een applet maken waarin de gebruiker een getal kan intypen; hierna wordt het getal + 1 getoond.

1. Klik links in de zogenaamde methodenlijst op *Initialiseren* zodat deze gemarkeerd wordt.
2. Klik rechts op de tab *GUI*.
3. Klik onder de tabbladen op *Maak een invoervak (TextField)*.
4. Voer in het venster dat nu verschijnt in: *tekstvakje* en *5*. Klik op *OK*.
5. Klik onder de tabbladen op *Maak een knop (Button)*.
6. Voer in het venster in: *knopje*, *"Klik hier"* en *berekenen*. Klik op *OK*.
7. Klik onder de tabbladen op *Maak een vast tekstvak (Label)*.
8. Voer in het venster in: *label* en *"Typ een getal in en klik op de knop"*. Klik op *OK*.
9. Als het goed is, ziet het vak onder *PSD:* er als volgt uit:

Maak het invoervak tekstvakje
Maak de knop knopje
Maak het vaste tekstvak label

10. Klik in de menubalk bovenin het scherm op *Methoden > Methode toevoegen…* Typ bij *Methodenaam* in: *berekenen*.
11. Vul de onderste van de twee tabellen in zoals hieronder aangegeven:



12. Laat alle andere velden zoals ze zijn. Klik op *OK*.
13. Klik links in het hoofdscherm in de methodenlijst op *berekenen* zodat deze gemarkeerd wordt.
14. Klik onder de tabbladen op *Zet de inhoud van een invoervak in een variabele*.
15. Voer in het venster in: *tekstvakje* en *vakjeTekst*. Klik op *OK*.
16. Klik rechts op de tab *Variabelen*.
17. Klik onder de tabbladen op *Verander een tekenreeks in een geheel getal*.
18. Voer in het venster in: *vakjeTekst* en *getal*. Klik op *OK*.
19. Klik rechts op de tab *GUI*.
20. Klik onder de tabbladen op *De tekst in een vast tekstvak aanpassen*.
21. Voer in het venster in: *label* en *"Uitkomst: " + (getal + 1)*. Klik op *OK*.
22. Als het goed is, ziet het vak onder *PSD:* er nu als volgt uit:

Zet inhoud van tekstvakje in vakjeTekst
Verander vakjeTekst in geheel getal getal
Tekst van label veranderen in "Uitkomst: " + (getal + 1)

23. Klik in de menubalk op *Uitvoeren > Java-code genereren…* Typ in het dialoogvenster een naam in voor het bestand. Achter de bestandsnaam moet u de extensie *.java* typen. Klik op *Save*.
24. Zoek op de harde schijf het gegenereerde bestand op en open het. In het bestand staat de Java-code voor de applet.
25. Als u de applet ook wilt uitvoeren, klikt u in het menu eerst op *Opties > Opties…* en vul de paden naar *javac* en *appletviewer* in (deze zijn geïnstalleerd bij de JDK). Klik daarna op *Uitvoeren > Java-code genereren en uitvoeren…* Geef weer de bestandsnaam in (met *.java*) en klik weer op *Save*. Als alles goed is gegaan, dan verschijnt na enkele seconden de applet in beeld.

Meer hulp nodig?
Komt u er niet uit? Open dan het programma en kies voor *Help > Help* in het menu. U vindt hier uitgebreide informatie over het programma en over methoden en PSD's. U kunt ook naar de website van het programma gaan: *http://home.kpn.nl/sonke005/pws/appletcreator.htm*. Daar vindt u extra hulp.

JAR-bestand maken

Dit gaat op dezelfde manier als bij de PSD Creator. Zie hiervoor pagina 34. Het gemaakte JAR-bestand is te vinden op de CD.

De website maken

De Applet Creator (en de PSD Creator) krijgen ook een website, waarop gebruikers het programma en de broncode kunnen downloaden en waar ze meer informatie en hulp kunnen krijgen. Dit blijft een vrij simpele website.

Als de gebruiker de site (*http://home.kpn.nl/sonke005/pws*) opstart, dan krijgt hij een introductie te zien waarbij hij kan kiezen uit *Applet Creator* of *PSD Creator*. Zie *Figuur 43*.

Figuur 43 – Het introductiescherm van de website.

De knoppen verwijzen naar twee framepagina's: *applet.htm* en *psd.htm*. In deze framepagina's staat aan de linkerkant een menu met dezelfde twee knoppen (maar wel kleiner). Aan de rechterkant van het venster staat in deze pagina's de echte inhoud: *applet_inh.htm* en *psd_inh.htm*.

Figuur 44 – De pagina over de Applet Creator op de website

Javadoc-documentatie genereren

Als laatste maak ik Javadoc-documentatie. Ik heb tijdens het maken van de code al Javadoc-commentaar toegevoegd, dus ik hoef alleen *javadoc.exe* te laten lopen. Hiervoor gebruik ik het volgende commando:

```
"C:\Program Files\Java\jdk1.6.0_16\bin\javadoc" -private -d javadoc -
verbose -linksource -windowtitle "Applet Creator Javadoc-documentatie" -
doctitle "Applet Creator Javadoc-documentatie" -header "Applet Creator
v1.0.0<br>Javadoc" -use -splitindex -link
http://java.sun.com/javase/1.6.0/docs/api *.java
```

Schakelopties van Javadoc

-private: genereer ook documentatie voor *private* en *package private* methoden en klassen.
-d javadoc: zet de documentatie in de map *javadoc*.
-linksource: maak HTML-bestanden met de broncode erin. Als er nu wordt geklikt op een methode- of klassenaam, wordt de broncode van dit element weergegeven.
-windowtitle "…": de titel van het browservenster.
-doctitle "…" dit is de titel van de documentatie.
-header "…": de tekst die rechtsboven in de documentatie komt te staan.
-use: genereer pagina's waarop wordt aangegeven door welke andere klassen iedere klasse gebruikt wordt. Klik in de navigatiebalk op *Use* om deze pagina's te zien.
-splitindex: maak voor iedere beginletter in de index een aparte pagina, zodat je minder hoeft te scrollen.
-link …: Deze optie krijgt de URL van de javadoc-documentatie van de Java API mee. De optie zorgt ervoor dat er in mijn eigen documentatie links worden gemaakt naar de pagina's uit de Java-documentatie, zodat je gemakkelijk informatie over elementen uit de Java API kunt opvragen.

De gegenereerde documentatie is weer te vinden op de CD; ook van de PSD Creator heb ik documentatie gemaakt.
Stap 6 is nu klaar.

Overzicht van de bestanden op de CD

Hieronder staat een overzicht van de bestanden die op de CD staan, geordend in de weergegeven mappenstructuur.

- Map *Applet Creator*
  - Een JAR-bestand om de Applet Creator eenvoudig te kunnen openen.
  - Een PDF-bestand met de handleiding van de Applet Creator.
  - Map *Code*
    - De broncode (.class- en .java-bestanden).
    - Map *help*
      - De helpbestanden voor het Helpvenster.
    - Map *images*
      - De iconen van het programma. Ook staat hierin het PowerPoint-bestand waarmee ik de iconen gemaakt heb.
    - Map *javadoc*
      - De javadoc-documentatie van de Applet Creator. Klik op *index.htm* om de documentatie te bekijken.
  - Map *Gemaakte programma's*
    - De programma's zoals die gegenereerd zijn door de Applet Creator in stap 5 (Testen), inclusief de HTML- en de gecompileerde .class-bestanden.
- Map *PSD Creator*
  - Een JAR-bestand om de PSD Creator eenvoudig te kunnen openen.
  - Een PDF-bestand met de handleiding van de PSD Creator.
  - Map *Code*
    - De broncode (.class- en .java-bestanden).
    - Map *images*
      - De iconen van het programma.
    - Map *javadoc*
      - De javadoc-documentatie van de PSD Creator. Klik op *index.htm* om de documentatie te bekijken.
- Map *referentieTest*
  - De broncode en de .class-bestanden voor het testprogrammaatje dat ik geschreven heb om te testen of Java alleen referenties meegeeft of de volledige objecten. Zie pagina 48.
- Map *Documentatie*
  - Documentatie.docx: dit documentatiebestand als Word 2007-bestand.
  - Documentatie.doc: dit documentatiebestand als Word 2003-bestand, mocht u het .docx-bestand niet kunnen lezen. Let op: ik heb het .doc-bestand niet gecontroleerd op fouten, dus misschien zijn er dingen verschoven.
  - Documentatie.pdf: dit documentatiebestand in PDF-formaat.
  - Planning.xlsx: planning en logboek van het project.

Evaluatie

Ik denk dat ik bij dit project heel veel geleerd heb, vooral over Java. Voordat ik aan het project (en aan Ockellaravity, zie bronnenlijst) begon wist ik bijvoorbeeld niets over het maken van Applications in plaats van Applets, over constructors, over typecasting, over polymorfisme, enzovoorts.
De planning had – zoals ik van mezelf eerlijk gezegd wel gewend ben – beter gekund: ik had maar 43 uur ingepland en uiteindelijk ben ik er zo'n 131 uur mee bezig. Het werk valt altijd langer uit dan gepland door bugs en andere problemen die opgelost moeten worden.
Over het product zelf ben ik redelijk tevreden: het is mogelijk om met het programma werkende applets te maken. Het probleem is echter dat het voor niet-Javakenners toch nog steeds een zware klus is om het programma te begrijpen, onder meer door het gebrek aan controle (je kunt zonder problemen een niet-bestaande variabele of iets dergelijks in een blok invullen, en bijvoorbeeld bij het intypen van een bestandsnaam word je er helemaal niet op gewezen dat die geen spaties mag bevatten). Bovendien is er geen mogelijkheid tot het opslaan en openen van bestanden. Misschien kan ik dit in volgende versies nog verbeteren.

Bronnenlijst

- The Java Tutorials (*http://java.sun.com/docs/books/tutorial*). Dit is een mooie website waarop ik vele oplossingen van problemen gevonden heb. Ook zijn sommige klassen gebaseerd op voorbeelden uit The Java Tutorials.
- Het boekje *Java: my cup of tea* van dhr. Stoop.
- De Java API-documentatie (*http://java.sun.com/javase/6/docs/api*).
- Het project Ockellaravity, dit is een project dat Laurey-Anne Krammer, Stijn Köster en ik maken bij het vak Informatica als eindproject. Het doel van het project is het maken van een zwaartekrachtsimulatie. Het helpvenster heb ik grotendeels met toestemming overgenomen uit Ockellaravity.
- *http://www.rgagnon.com/javadetails/java-0058.html*: om problemen op te lossen bij de uitvoer.
- *http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-traps.html*: een artikel over het starten van externe programma's vanuit Java. Dit heb ik gebruikt om *javac* en *appletviewer* te starten vanuit het programma.